# Real-Time Workshop®

For Use with SIMULINK®

**Modeling**

**Simulation**

**Implementation**

User's Guide

*Version 2.1*

**How to Contact The MathWorks:**

| | |
|---|---|
| 508-647-7000 | Phone |
| 508-647-7001 | Fax |
| The MathWorks, Inc.<br>24 Prime Park Way<br>Natick, MA 01760-1500 | Mail |
| `http://www.mathworks.com` | Web |
| `ftp.mathworks.com` | Anonymous FTP server |
| `comp.soft-sys.matlab` | Newsgroup |
| `support@mathworks.com` | Technical support |
| `suggest@mathworks.com` | Product enhancement suggestions |
| `bugs@mathworks.com` | Bug reports |
| `doc@mathworks.com` | Documentation error reports |
| `subscribe@mathworks.com` | Subscribing user registration |
| `service@mathworks.com` | Order status, license renewals, passcodes |
| `info@mathworks.com` | Sales, pricing, and general information |

*Real-Time Workshop User's Guide*

| | | | |
|---|---|---|---|
| Printing History: | May 1994 | First printing | Version 1.0 |
| | May 1997 | Second printing | Version 2.1 |
| | January 1998 | Revised for 2.2 | (Online version) |

# Contents

# 2 Getting Started with the Real-Time Workshop

# 5 Model Code

# 6 Program Architecture

# 7 Models With Multiple Sample Rates

# 9 Targeting DOS for Real-Time Applications

# 10 Targeting Custom Hardware

# 11 RTWlib

# A Real-Time Workshop Directory Tree

# Before You Begin

## Related Products and Documentation

### Requirements

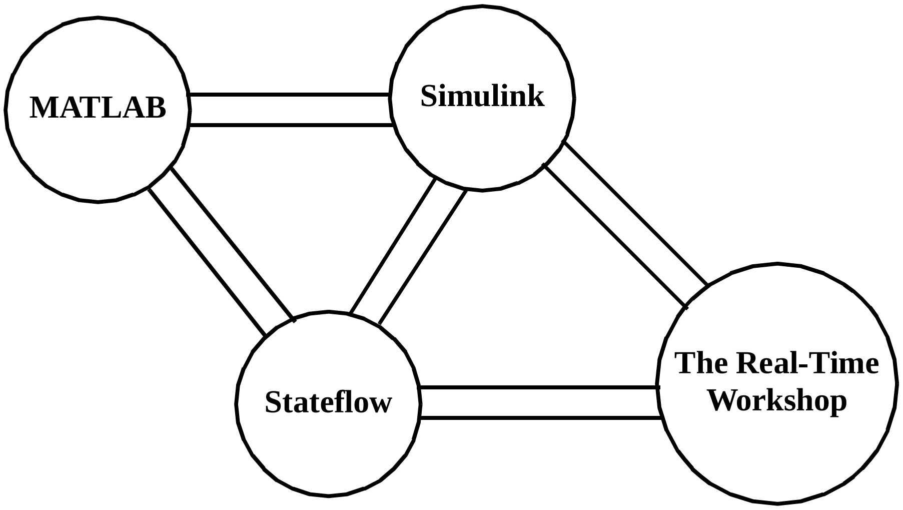
The Real-Time Workshop® is a multiplatform product, running on Microsoft Windows95, Windows NT, and UNIX systems.

The Real-Time Workshop requires:

- MATLAB® 5.1
- Simulink® 2.1
- Visual C++ or Watcom C language compiler

See the *MATLAB Application Program Interface Guide* for information on how to ensure proper installation for integration into the MATLAB environment.

In addition, you can optionally use the Stateflow™ product to add finite-state machine modules to your model. The Real-Time Workshop generates production quality code for models that consist of elements from MATLAB, Simulink, and Stateflow:



## What Is MATLAB?

MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include:

- Math and computation
- Algorithm development
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including graphical user interface building

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a scalar noninteractive language such as C or Fortran.

The name MATLAB stands for *matrix laboratory*. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects, which together represent the state-of-the-art in software for matrix computation.

MATLAB has evolved over a period of years with input from many users. In university environments, it is the standard instructional tool for introductory and advanced courses in mathematics, engineering, and science. In industry, MATLAB is the tool of choice for high-productivity research, development, and analysis.

MATLAB features a family of application-specific solutions called *toolboxes*. Very important to most users of MATLAB, toolboxes allow you to *learn* and *apply* specialized technology. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many others.

See the MATLAB documentation set for more information.

## What Is Simulink?

Simulink is a software package for modeling, simulating, and analyzing dynamic systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e., have different parts that are sampled or updated at different rates.

For modeling, Simulink provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations. With this interface, you can draw the models just as you would with pencil and paper (or as most textbooks depict them). This is a far cry from previous simulation packages that require you to formulate differential equations and difference equations in a language or program. Simulink includes a comprehensive block library of sinks, sources, linear and nonlinear components, and connectors. You can also customize and create your own blocks.

Models are hierarchical, so you can build models using both top-down and bottom-up approaches. You can view the system at a high-level, then double-click on blocks to go down through the levels to see increasing levels of model detail. This approach provides insight into how a model is organized and how its parts interact.

After you define a model, you can simulate it, using a choice of integration methods, either from the Simulink menus or by entering commands in MATLAB's command window. The menus are particularly convenient for interactive work, while the command-line approach is very useful for running a batch of simulations (for example, if you are doing Monte Carlo simulations or want to sweep a parameter across a range of values). Using scopes and other display blocks, you can see the simulation results while the simulation is running. In addition, you can change parameters and immediately see what happens, for “what if” exploration. The simulation results can be put in the MATLAB workspace for postprocessing and visualization.

Model analysis tools include linearization and trimming tools, which can be accessed from the MATLAB command line, plus the many tools in MATLAB and its application toolboxes. And because MATLAB and Simulink are integrated, you can simulate, analyze, and revise your models in either environment at any point.

*Using Simulink* describes how to work with Simulink. It explains how to manipulate Simulink blocks, access block parameters, and connect blocks to build models. It also provides reference descriptions of each block in the standard Simulink libraries.

## What Is Stateflow?

Stateflow is a powerful graphical design and development tool for complex control and supervisory logic problems. Stateflow supports flow diagram notation as well as state transition notation. Using Stateflow you can:

- Visually model and simulate complex reactive systems based on *finite state machine* theory.
- Design and develop deterministic, supervisory control systems.
- Use flow diagram notation and state transition notation seamlessly in the same Stateflow diagram.
- Easily modify your design, evaluate the results, and verify the system's behavior at any stage of your design.
- Automatically generate integer or floating-point code directly from your design.
- Take advantage of the integration with the MATLAB and Simulink environments to model, simulate, and analyze your system.

Flow diagram notation is essentially logic represented without the use of states. In some cases, using flow diagram notation is a closer representation of the system’s logic and avoids the use of unnecessary states. Flow diagram notation is an effective way to represent common code structures like `for` loops and `if-then-else` constructs.

Stateflow also provides clear, concise descriptions of complex system behavior using finite state machine theory, flow diagram notations, and state-transition diagrams. Stateflow brings system specification and design closer together. It is easy to create designs, consider various scenarios, and iterate until the Stateflow diagram models the desired behavior.

### Using Stateflow with Simulink

Stateflow is part of the Simulink environment. The Stateflow block is a masked Simulink model. Stateflow builds an S-function that corresponds to each Stateflow machine. This S-function is the agent Simulink interacts with for simulation and analysis.

The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow diagrams into Simulink models, you can add event-driven behavior to Simulink simulations. You create models that represent both data and control flow by combining Stateflow blocks with the standard Simulink blocksets. These combined models are simulated using Simulink.

## Using Stateflow with the Real-Time Workshop

Stateflow works in conjunction with Real-Time Workshop in these ways:

- Stateflow generates high quality integer-based code (runs on a fixed-point processor; a floating-point processor is not required).
- The Real-Time Workshop generates floating-point-based code (a floating-point processor is required).
- Stateflow blocks are fully integrated for real-time code generation and compatible with the Real-Time Workshop.

---

**Note:** If you want to target Stateflow and the Real-Time Workshop generated code for the same processor, you need a floating-point processor.

---

# How to Use This Guide

## Typographical Conventions

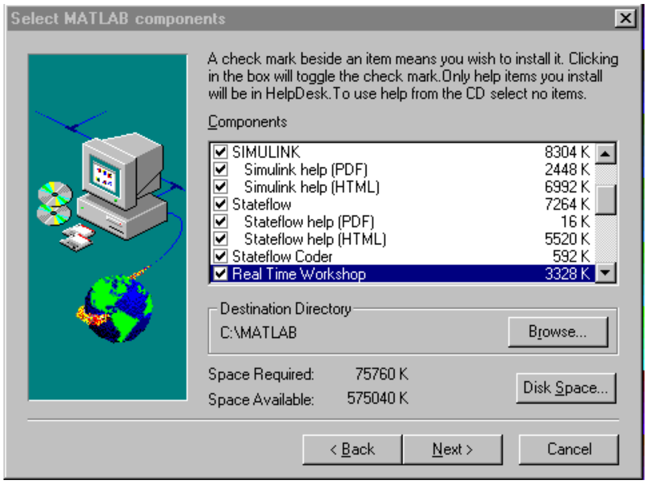
| To Indicate | This Guide Uses | Example |
|---|---|---|
| Example code | `Monospace type` | To assign the value 5 to A, enter<br><br>`A = 5` |
| Function names | `Monospace type` | The `cos` function finds the cosine of each array element. |
| Function syntax | `Monospace type` for text that must appear as shown.<br><br>*`Monospace italics`* for components you can replace with any variable. | The `magic` function uses the syntax<br><br>*M* = `magic(`*n*`)` |
| MATLAB output | `Monospace type` | MATLAB responds with<br><br>`A =`<br><br>`    5` |

## Installation

Your platform-specific MATLAB *Installation Guide* provides essentially all of the information you need to install the Real-TimeWorkshop.

Prior to installing the Real-Time Workshop, you must obtain a License File or Personal License Password from The MathWorks. The License File or Personal License Password identifies the products you are permitted to install and use.

As the installation process proceeds, a screen similar to this, where you can indicate which products to install, is displayed:



Select MATLAB components

A check mark beside an item means you wish to install it. Clicking in the box will toggle the check mark. Only help items you install will be in HelpDesk. To use help from the CD select no items.

Components

| | |
|---|---|
| ☑ SIMULINK | 8304 K |
| ☑ Simulink help (PDF) | 2448 K |
| ☑ Simulink help (HTML) | 6992 K |
| ☑ Stateflow | 7264 K |
| ☑ Stateflow help (PDF) | 16 K |
| ☑ Stateflow help (HTML) | 5520 K |
| ☑ Stateflow Coder | 592 K |
| ☑ Real Time Workshop | 3328 K |

Destination Directory: C:\MATLAB

Space Required: 75760 K
Space Available: 575040 K

The Real-Time Workshop has certain product prerequisites that must be met for proper installation and execution:

| Licensed Product | Prerequisite Products | Additional Information |
|---|---|---|
| Simulink 2.2 | MATLAB 5.2 | Allows installation of Simulink. |
| The Real-Time Workshop | Simulink 2.2 | Requires a Watcom C or Visual C++ compiler for creating MATLAB MEX-files on your platform. |

If you experience installation difficulties and have Web access, connect to The MathWorks home page (`http://www.mathworks.com`). Look for the license manager and installation information under the `Tech Notes/FAQ` link under `Tech Support Info`.

# Introduction to the Real-Time Workshop

## Introduction

The Real-Time Workshop®, for use with MATLAB® and Simulink®, produces code directly from Simulink models and automatically builds programs that can be run in a variety of environments, including real-time systems and stand-alone simulations.

With the Real-Time Workshop, you can run your Simulink model in real-time on a remote processor. The Real-Time Workshop also enables you to run high-speed stand-alone simulations on your host machine or on an external computer.

Using the rapid prototyping process, you can shorten development cycles and reduce costs. You can also use the Real-Time Workshop to implement hardware-in-the-loop (HIL) simulations.

This chapter presents an introduction to the Real-Time Workshop and describes its relationship to Simulink and MATLAB. The chapter concludes with a summary of the material contained in this manual and a description of the examples that are bundled with the product.

## The Real-Time Workshop

The Real-Time Workshop provides a real-time development environment that features:

- A rapid and direct path from system design to hardware implementation.
- Seamless integration with MATLAB and Simulink.
- A simple, easy to use interface.
- An open and extensible architecture.
- A fully configurable code generator — every aspect of the generated code can be configured by using the Target Language Compiler™.
- Fast design iterations by editing block diagrams and automatically building a new executable.

The package includes application modules that allow you to build complete programs targeting a wide variety of environments. Program building is fully automated.

### Real-Time Workshop Applications

The Real-Time Workshop is designed for a variety of applications. Some examples include:

- Real-time control — You can design your control system using MATLAB and Simulink and generate code from your block diagram model. You can then compile and download it directly to your target hardware.
- Real-time signal processing — You can design your signal processing algorithm using MATLAB and Simulink. The generated code from your block diagram can then be compiled and downloaded to your target hardware.
- Hardware-in-the-loop simulation — You can create Simulink models that mimic real-life measurement, system dynamics, and actuation signals. Generated code from the model can be targeted on special purpose hardware to provide a real-time representation of the physical system. Applications

include control system validation, training simulation, and fatigue testing using simulated load variations.

- Interactive real-time parameter tuning — You can use Simulink as a front end to your real-time program. This allows you to change parameters while the program is executing.
- High-speed stand-alone simulations.
- Generation of portable C code for export to other simulation programs.

## The Generated Code

The generated code (i.e., the model code) is by default highly optimized and fully commented C code that can be generated from any Simulink model, including linear, nonlinear, continuous, discrete, or hybrid models.

All Simulink blocks are automatically converted to code, with the exception of MATLAB function blocks and S-function blocks that invoke M-files. You must rewrite these blocks as C MEX S-functions if you want to use them with the Real-Time Workshop.

The Real-Time Workshop includes a set of target files that are compiled by the Target Language Compiler (TLC) to produce ANSI C code. The target files are ASCII text files that describe how to convert the Simulink model to code. For advanced users, target files enable you to customize generated code for individual blocks or throughout the entire model. See the *Target Language Compiler Reference Guide* for more information about customizing target files.

You can incorporate C MEX S-functions, along with the generated code, into the program executable. You can also write a target file for your C MEX S-function to *inline* the S-function, thus improving performance by eliminating function calls to the S-function itself.

### Types of Output

The Real-Time Workshop's interface allows you to select three forms of output:

- C code — Generate code that contains system equations and initialization functions for the Simulink model. You can use this code in nonreal-time simulation environments or for real-time applications.
- A real-time program — Transform the generated code into a real-time program suitable for use with dedicated real-time hardware. The resulting code is designed to interface with an external clock source and hence runs at

a fixed, user-specified sample rate. Continuous time models are incorporated into this code with the simple provision that their states are propagated using fixed-step size integration algorithms.

- A high-performance stand-alone simulation — use the generated code with the generic real-time system target file to produce an executable for stand-alone simulations. At the end of the simulation, the executable produces a `model.mat` file that contains the MATLAB variables that were logged in Simulink. This `.mat` file is used for analysis in MATLAB.

  You can also use the stand-alone generic real-time simulation environment for code validation since you can directly compare results against Simulink.

## The Rapid Prototyping Process

The Real Time Workshop allows you to do *rapid prototyping*, a process that allows you to conceptualize solutions using a block diagram modeling environment and take an early look at system performance prior to laying out hardware, writing any production software, or committing to a fixed design.

By using rapid prototyping, you can refine your real-time system by continuously iterating your model. Further, you can tune parameters on the fly by using Simulink as the front-end of your real-time model. This is known as using Simulink in *external mode*.

### Key Aspects of Rapid Prototyping

The key to rapid prototyping is automatic code generation. It reduces algorithm coding to an automated process; this includes coding, compiling, linking, and downloading to target hardware. This automation allows design changes to be made directly to the block diagram. This figure show the rapid prototyping development process:

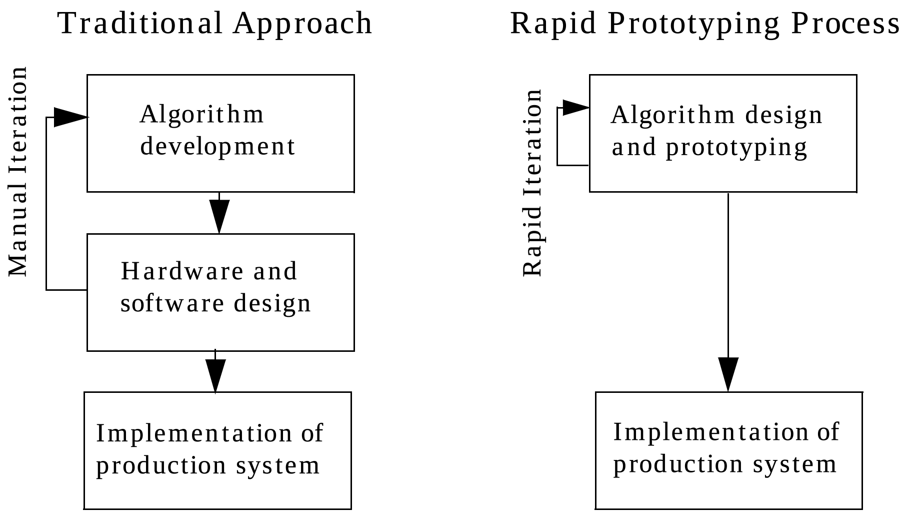

Traditional Approach
Rapid Prototyping Process
Manual Iteration
Algorithm development
Hardware and software design
Implementation of production system
Rapid Iteration
Algorithm design and prototyping
Implementation of production system

**Figure 1-1: Comparison of Traditional and Rapid Prototyping Development Processes**

The traditional approach to real-time design and implementation typically involves multiple teams of engineers, including an algorithm design team, software design team, hardware design team, and an implementation team. When the algorithm design team has completed its specifications, the software design team implements the algorithm in a simulation environment and then specifies the hardware requirements. The hardware design team then creates the production hardware. Finally, the implementation team integrates the hardware into the larger overall system. This approach leads to long development processes, because the algorithm design engineers do not work with the actual hardware. The rapid prototyping process combines the algorithm, software, and hardware design phases, thus eliminating potential bottlenecks. The process allows engineers to see the results and rapidly iterate on the design before expensive hardware is developed.

The rapid prototyping process begins in Simulink. First, you develop a model in Simulink. In control engineering, this involves modeling plant dynamics. In digital signal processing, the model is typically an exploration of the signal-to-noise ratio and other characteristics of the input signal. You then run your model in Simulink; you can use MATLAB, Simulink, and toolboxes to develop algorithms and analyze the results. If the results are not satisfactory, you can iterate the modeling/analysis process until results are acceptable.

Once you have achieved the desired results, you can use the Real-Time Workshop to generate downloadable C code. Using Simulink in external mode, you can tune parameters and further refine your model, again rapidly iterating to achieve required results. Finally, the code is ready for use in production systems.

This figure shows the rapid prototyping process in more detail:

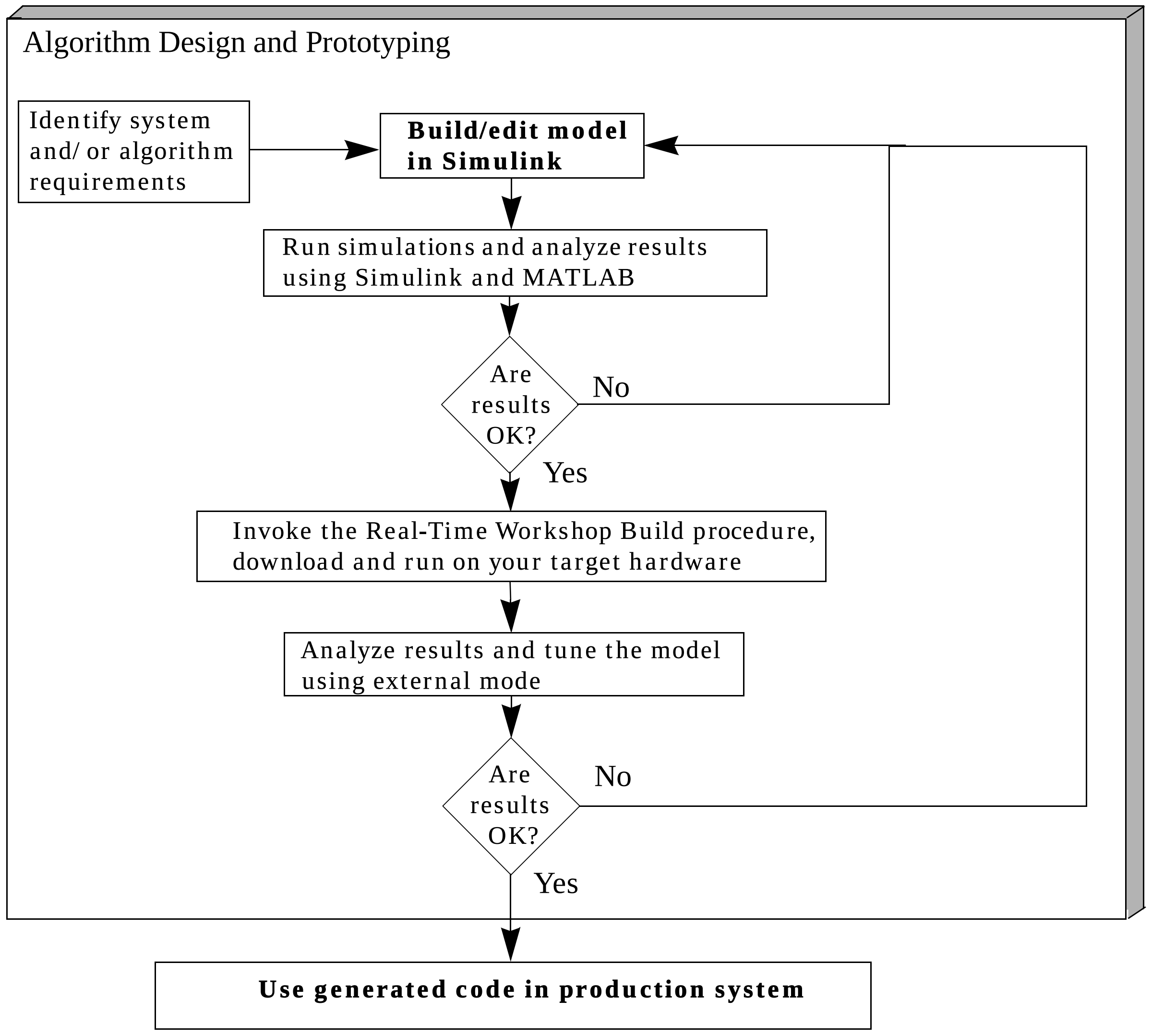

Algorithm Design and Prototyping
Identify system and/ or algorithm requirements
Build/edit model in Simulink
Run simulations and analyze results using Simulink and MATLAB
Are results OK?
No
Yes
Invoke the Real-Time Workshop Build procedure, download and run on your target hardware
Analyze results and tune the model using external mode
Are results OK?
No
Yes
Use generated code in production system

**Figure 1-2: The Rapid Prototyping Development Process**

This highly productive development cycle is possible because the Real-Time Workshop is closely tied to MATLAB and Simulink. Each package contributes to the design of your application:

- MATLAB — Design, analysis, and data visualization tools
- Simulink — System modeling, simulation, and validation
- Real-Time Workshop — C code generation from Simulink model and framework for running generated code in real-time

## Rapid Prototyping for Digital Signal Processing

The first step in the rapid prototyping process for digital signal processing is to consider the kind and quality of the data to be worked on and to relate it to the system requirements. Typically this includes examining the signal-to-noise ratio, distortion, and other characteristics of the incoming signal, and relating them to algorithm and design choices.

### System Simulation and Algorithm Design

In the rapid prototyping process, the role of the block diagram in algorithm development is twofold. It supplies a way to identify processing bottlenecks and to optimize the algorithm or system architecture. It also provides a high-level system description, that is, a hierarchical framework for evaluating the behavior and accuracy of alternative algorithms under a range of operating conditions.

### Analyzing Results and Parameter Tuning Using External Mode

Once an algorithm (or a set of candidate algorithms) has been created, the next stage is to consider architectural and implementation issues such as complexity, speed, and accuracy. In the conventional case, this meant recoding the algorithm in C or in a hardware design and simulation package.

After building the executable and downloading it to your hardware, you can modify block parameters in Simulink and automatically download the new values to the hardware. To change these parameters from the Simulink block diagram, you can run Simulink in external mode using the Simulink pull-down menu. Simulink's external mode allows you to change parameters interactively without stopping the real-time execution of your signal processing algorithms on the hardware.

## Rapid Prototyping for Control Systems

Rapid prototyping for control systems is similar to digital signal processing, with one exception: in control systems design, it is necessary to develop a model of your plant prior to algorithm development. Once a simulation has been developed that models the plant with sufficient accuracy, the rapid prototyping process for control system design continues in much the same manner as digital signal processing design.

Rapid prototyping begins with developing block diagram plant models of sufficient fidelity for preliminary system design and simulation. Once simulations show encouraging system performance, the controller block diagram is separated from the plant mode and I/O device drivers are attached. Using automatic code generation, the entire system is immediately converted to real-time executable code. The executable can be automatically loaded onto target hardware, allowing the implementation of real-time control systems in very short time.

### Modeling Systems in Simulink

You can use MATLAB and Simulink to design, test, and analyze a model of your system. The first step in the design process is to develop a plant model. You can build models involving plant, sensor, and actuator dynamics using the Simulink collection of linear and nonlinear components. Because Simulink is customizable, you can further simplify modeling by creating custom blocks and block libraries from continuous and discrete-time components.

You can also use the Systems Identification Toolbox to analyze test data to develop an empirical plant model, or the Symbolic Math Toolbox to translate the equations of the plant dynamics into state-variable form.

### Analyze Results of the Simulation

You can use MATLAB and Simulink to analyze the results produced from the model you developed in the first step of the rapid prototyping process. It is at this stage that you can design and add a controller to your plant.

### Algorithm Design and Analysis

From the block diagrams developed during the modeling stage, you can extract state-space models through linearization techniques. These matrices can be used in control system design. You can use these toolboxes to facilitate control system design, and work with the matrices that you derived:

- Control System Toolbox
- Robust Control Toolbox
- Model Predictive Control Toolbox
- μ-Analysis and Synthesis Toolbox
- LMI Control Toolbox
- QFT Control Toolbox

Once you have your controller designed, you can create a closed-loop system by connecting it to the Simulink plant model. Closed-loop simulations allow you to determine how well the initial design meets performance requirements.

Once you have a model that you're satisfied with, it is a simple matter to generate C code directly from the Simulink block diagram, compile it for the target processor, and link it with supplied or user-written application modules.

### Analyzing Results and Parameter Tuning Using External Mode

You can load the program data into MATLAB for analysis or display the data with third party monitoring tools. You can easily make design changes to the Simulink model and then regenerate the C code.

After building the executable and downloading it to your hardware, you can modify block parameters in Simulink and automatically download the new values to the hardware. To change these parameters from the Simulink block diagram, you can run Simulink in external mode. Simulink's external mode allows you to change parameters interactively without stopping the real-time execution of the model on the hardware.

## Open Architecture of the Real-Time Workshop

The Real-Time Workshop is an open and extensible system designed for use with a wide variety of operating environments and hardware types. Its features include:

- The ability to generate code from any fixed-step Simulink block diagram
- A framework for building real-time programs
- An extensible device driver library supporting a variety of hardware
- Automatic program building and a fully customizable build process
- Bundled sample implementations in DOS, Tornado, and generic real-time environments
- Support for third-party hardware and tools
- Fully customizable code generation, including inlined custom blocks
- Automatic loop-rolling, which allows vectorized operations to be expanded out or placed in a “for” loop, depending upon a rolling threshold that you can set
- The ability to configure the build process to virtually any version of `make`. The Real Time Workshop includes examples that use Watcom C/C++, Microsoft Visual C/C++, and UNIX versions of `make`.

You can customize the code generation and build process of the Real-Time Workshop. The following picture highlights the open architecture of the Real-Time Workshop. The next few pages discuss concepts in this figure:

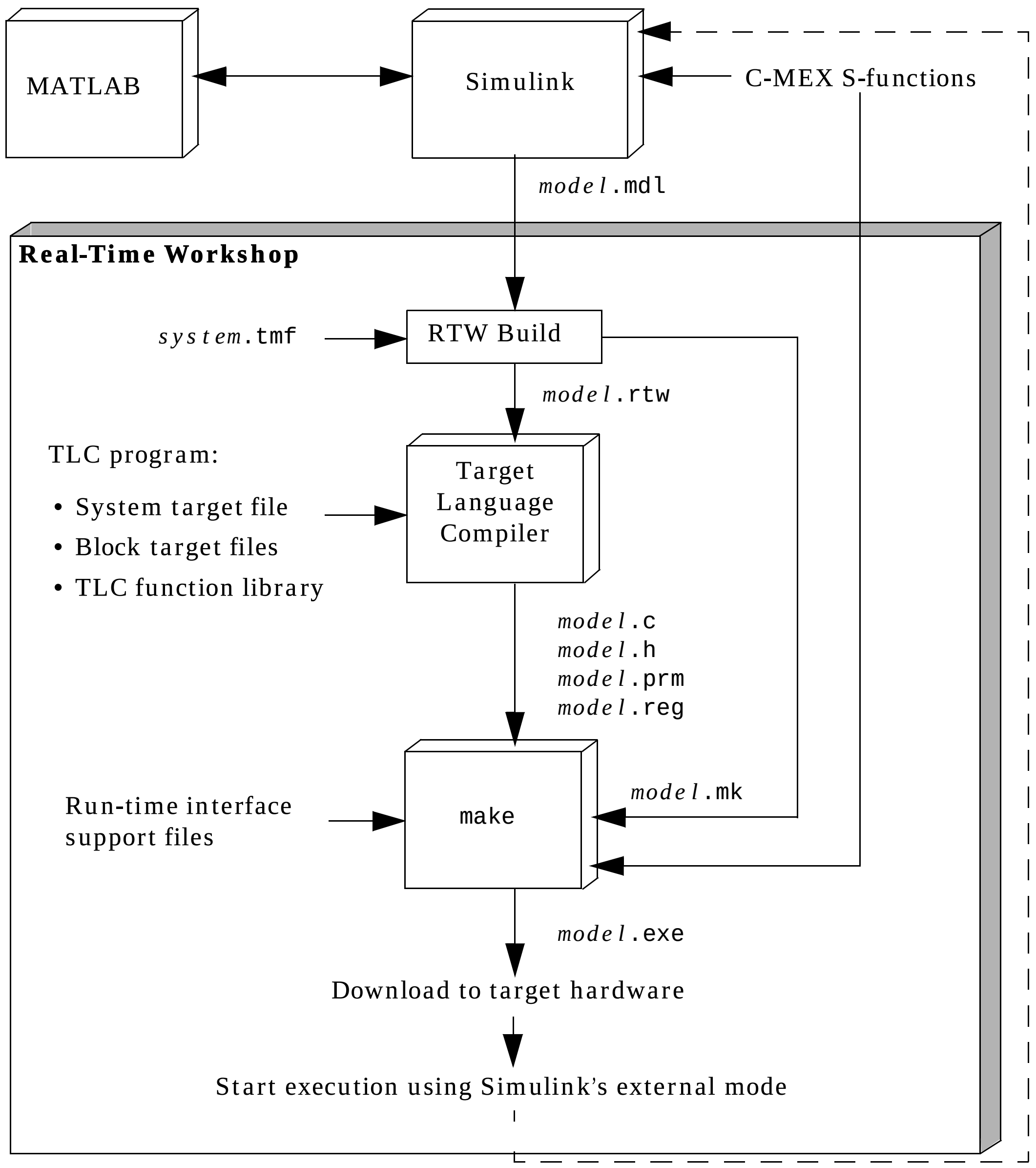


**Figure 1-3: The Real-Time Workshop's Open Architecture**

### Target Language Compiler

To generate code, the Real-Time Workshop invokes the Target Language Compiler (TLC). The TLC transforms a description generated by the Real-Time Workshop of your Simulink model into target specific code. The description of your model is saved in an ASCII file called *model*`.rtw`.

The Target Language Compiler allows you to modify most aspects of the generated code. The compiler reads the *model*`.rtw` file and executes a TLC program that consists of a set of *target files* (`.tlc` files). These are ASCII files written for the Target Language Compiler. The *TLC program* specifies how to transform the model.rtw file into generated code.

The TLC program consists of:

- The entry point or main file. This is called the *system target file*.
- A set of *block target files*. These specify how to translate each block in your model into target-specific code.
- A *TLC function library*. This is a set of library functions that the TLC program uses when converting the *model*`.rtw` file into generated code.

The complete TLC program is provided with the Real-Time Workshop.

If you are familiar with HTML, Perl, and MATLAB, you may see similarities between these and the Target Language Compiler. The TLC has the mark-up language features, like HTML, the power and flexibility of Perl and other scripting languages, and the data handling power of MATLAB. The Target Language Compiler is designed for one purpose — to convert the model description file, *model*`.rtw`, into target-specific code.

### Make Utility

The Real-Time Workshop invokes `make` to build the real-time executable. `Make` is a utility that compiles and links the generated code to create an executable. The *model*`.mk` makefile, which the *system*`.tmf` (system template makefile) creates, in turn invokes `make`. You can fully configure the `make` utility by modifying the template makefile.

### S-Functions

S-functions allow you to add custom code to your Simulink model. You can embed the code directly into the generated code or, by default, allow the S-function Application Program Interface (API) to call the S-function.

Embedding the S-function code directly into the generated code is called *inlining* the S-function. The default is called *noninlined*.

For more information on S-functions, see *Using Simulink*. For information on inlining S-functions see the *Target Language Compiler Reference Guide*.

The Target Language Compiler is the key for customizing the generated code. Customizations usually involve inlining S-functions.

## The Build Procedure

The open and modular structure of the Real-Time Workshop allows you to configure the Real-Time Workshop for your environment, or use an existing environment, such as dSPACE or Wind River Systems Tornado Development Environment, provided by a third-party vendor.

The Real-Time Workshop build procedure first creates the *model*`.rtw` file, which is an intermediate representation of a Simulink block diagram that contains information such as the parameter values, vector widths, sample times, and execution order for the blocks in your model. This information is stored in a language-independent format.

After creating the *model*`.rtw` file, the build procedure invokes the Target Language Compiler to transform it into target-specific code. The Target Language Compiler begins by reading in the *model*`.rtw` file. It then compiles and executes the commands in the target files. The target files (sometimes referred to as `.tlc` files) specify how to transform the *model*`.rtw` file into target-specific code. The Target Language Compiler starts execution with the system target file. It then loads the individual block target files to transform the block information in the *model*`.rtw` file into target-specific code for the blocks. The output of the Target Language Compiler is a source code version of the Simulink block diagram.

The Target Language Compiler includes a TLC function library, which is a set of routines for use by the various target files.

The next step of the build procedure is to create a system makefile (*system*`.mk`) from a template makefile (*system*`.tmf`). *system*`.tmf` is the template makefile for your target environment; it allows you to specify compilers, compiler options, and additional information for the destination (target) of the generated executable.

The `system.mk` file is created by copying the contents of *system*`.tmf` and expanding out tokens describing your model's configuration. You can fully customize your build process by modifying an existing template makefile or providing your own template makefile.

After the *system*.mk file is created, the make command is invoked to create your executable. make can also optionally download the executable to your target hardware.

After downloading the file to the target hardware, if you are using external mode, you can connect back with Simulink to tune your model's parameters while the code is running on your target hardware.

### Files Created by the Build Procedure

Each of the *model*.* files performs a specific function in the Real-Time Workshop:

- *model*.mdl, created by Simulink, is analogous to a high-level programming language source file
- *model*.rtw, created by the RTW build process, is analogous to the object file created from a high-level language
- *model*.c, created by the TLC, is the C source code corresponding to the model.mdl file
- *model*.h, created by the TLC, is a header file that maps the links between blocks in the model
- *model*.prm, created by the TLC, contains the parameter settings of the blocks in the model
- *model*.reg, created by the TLC, contains the model registration function responsible for model initialization

## A First Look at Real-Time Workshop Files

An example of a Simulink model is this block diagram:

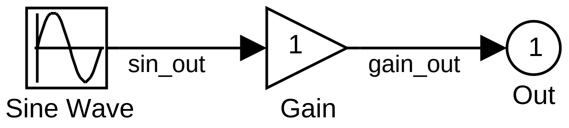


**Figure 1-4: A Simple Simulink Model**

This model is saved in a file called `example.mdl`. Generating C code from `example.mdl` is done by using the Real-Time Workshop user interface for generating code and executables. The next chapter, "Getting Started with the Real-Time Workshop," explains the details of the user interface.

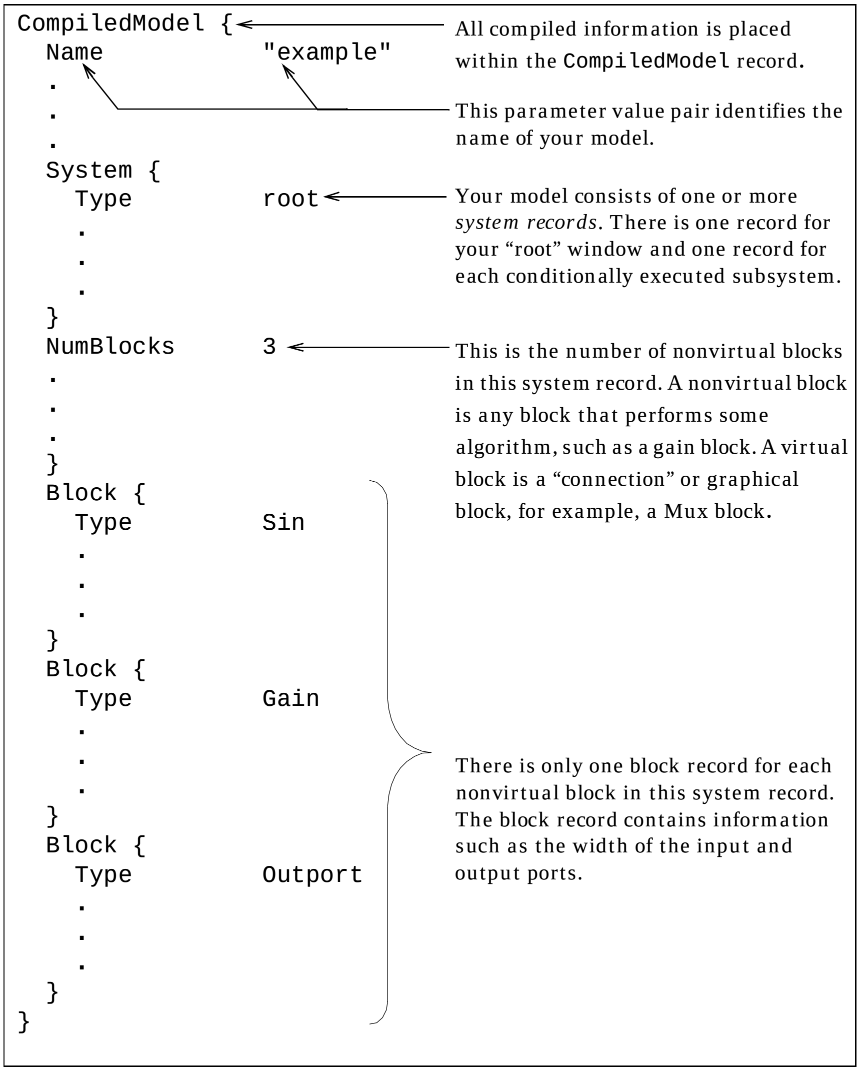
Later in this section are excerpts from the associated `.rtw` files that the Real-Time Workshop uses to generate the real-time version of this model in C.

### Basic Features of example.rtw

When you invoke the Real-Time Workshop build procedure to generate code, RTW first compiles your model. This compilation process consists of these tasks:

- Evaluating simulation and block parameters
- Propagating signal width and sample times
- Computing work vector sizes such as those used by S-functions (for more information about work vectors, refer to the Simulink documentation)
- Determining the execution order of blocks within the model

The Real-Time Workshop writes this information out to `example.rtw`. The `example.rtw` file is an ASCII file consisting of parameter value pairs stored in a hierarchical structure consisting of records. Below is an excerpt from example.rtw, which is the .rtw file associated with Figure 1-4: A Simple Simulink Model, on page 1-16:



```
CompiledModel {
  Name              "example"
  .
  .
  .
  System {
    Type            root
    .
    .
    .
  }
  NumBlocks         3
  .
  .
  .
  }
  Block {
    Type            Sin
    .
    .
    .
  }
  Block {
    Type            Gain
    .
    .
    .
  }
  Block {
    Type            Outport
    .
    .
    .
  }
}
```

All compiled information is placed within the CompiledModel record.

This parameter value pair identifies the name of your model.

Your model consists of one or more *system records*. There is one record for your “root” window and one record for each conditionally executed subsystem.

This is the number of nonvirtual blocks in this system record. A nonvirtual block is any block that performs some algorithm, such as a gain block. A virtual block is a “connection” or graphical block, for example, a Mux block.

There is only one block record for each nonvirtual block in this system record. The block record contains information such as the width of the input and output ports.

Note that *model*.rtw files are similar in appearance to *model*.mdl files generated by Simulink. For more information on .rtw files, see the *Target*

*Language Compiler Reference Guide*, which contains detailed descriptions of the contents of *model*.rtw files.

## Basic Features of example.c

Using the example.rtw file and target files, the Target Language Compiler creates C code that you can use in stand-alone or real-time applications. The generated C code consists of procedures that must be called by your target's execution engine.

These procedures consist of the algorithms defined by your Simulink block diagram. The execution engine executes the procedures as time moves forward. In addition to executing the generated procedures, your target may provide capabilities such as data logging. The modules that implement the execution engine and other capabilities are referred to collectively as the *run-time interface modules*.

For our example, the generated "Outputs" function, which must be called at every time step by the run-time interface, is shown below:

```
void MdlOutputs(int_T tid)
{
  /* Sin Block: <Root>/Sine Wave */
  rtB.sin_out = rtP.Sine_Wave.Amplitude *
  sin(rtP.Sine_Wave.Frequency*ssGetT(rtS) +
rtP.Sine_Wave.Phase);

  /* Gain Block: <Root>/Gain */
  rtB.gain_out = rtB.sin_out * rtP.Gain.Gain;

  /* Outport Block: <Root>/Out */
  rtY.Out = rtB.gain_out;
```

Note that the labels sin_out and gain_out that were placed in the model depicted in Figure 1-4 appear in the code of MdlOutputs. For further information on the contents of *model*.c files, refer to Chapter 3, "Code Generation and the Build Process,"

## Where to Go from Here

Chapter 2, “Getting Started with the Real-Time Workshop,” shows how to get started using the Real-Time Workshop, including how to build a generic real-time model. Third party environments (dSPACE, for example) are also discussed.

Chapter 3, “Code Generation and the Build Process,” describes the automatic program building process in detail, including a discussion of template makefiles.

Chapter 4, “External Mode, Data Logging, and Signal Monitoring,” contains information about external mode, parameter tuning, and data logging.

Chapter 5, “Model Code,” contains an example of model code.

Chapter 6, “Program Architecture,”discusses program architecture, including real-time program architecture, and the run-time interface.

Chapter 7, “Models With Multiple Sample Rates,” describes how to handle multirate systems.

Chapter 8, “Targeting Tornado for Real-Time Applications,” contains information that is specific to developing programs that target Tornado.

Chapter 9, “Targeting DOS for Real-Time Applications,” contains information that is specific to developing programs that target DOS.

Chapter 10, “Targeting Custom Hardware,” discusses the targeting of custom hardware, including the use of S-functions as device drivers.

Chapter 11, “RTWlib,” discusses how to create Interrupt Service Routines (ISRs) with the VxWorks Tornado system, how to add custom code to the RTW generated code, and how to create a customized asynchronous library for your target system.

Appendix A lists the directory structure and the files shipped with the Real-Time Workshop.

Appendix B is a glossary that contains definitions of terminology associated with the Real-Time Workshop and real-time development.

**Note**: See the *Target Language Compiler Reference Guide* for information about TLC files, including a discussion of custom target files.

## Bundled Target Systems

The Real-Time Workshop provides sample implementations that illustrate the development of real-time programs under DOS and Tornado, as well as generic real-time programs under Windows95, Windows NT, and UNIX.

These sample implementations are located in

- *matlabroot*`/rtw/c/grt` — Generic real-time examples
- *matlabroot*`/rtw/c/dos` — DOS examples
- *matlabroot*`/rtw/c/tornado` — Tornado examples

In addition to bundled systems provided by The MathWorks, third party vendors such as dSPACE provide real-time targets for a wide variety of rapid prototyping environments. For an example of a dSPACE implementation, see "Building a Real-Time Executable with dSPACE's RTI" on page 2–22.

## Using Stateflow and Blocksets with the Real-Time Workshop

Stateflow is a graphical design and development tool for complex control and supervisory logic problems. It supports flow diagram notation as well as state transition notation. Stateflow works seamlessly with the Real-Time Workshop. You can include Stateflow blocks in your Simulink model; the Real-Time Workshop generates code from them in much the same manner as it does for Simulink blocks and S-functions.

These blocksets are compatible with the Real-Time Workshop:

- DSP Blockset
- The Communications Toolbox — all Simulink blocks in the Toolbox
- Fixed Point Blockset

# Getting Started with the Real-Time Workshop

## Introduction

You can use the Real-Time Workshop in two stages. The first stage is to create a *generic real-time* executable from your model. This stand-alone executable runs on your workstation. It allows you to exercise and validate the generated code by simulating in single or multitasking environments.

The second stage is to use the generated code in a rapid prototyping environment such as the dSPACE, Tornado, or DOS real-time environments. Alternatively, you can target custom or “in-house” hardware. To do this you might want to start with the generic real-time target and modify the run-time interface files, the template makefile and/or any Target Language Compiler (TLC) files.

This chapter begins a discussion of basic concepts used in the Real-Time Workshop, such as generic real-time, targeting, and using template makefiles.

After discussing these concepts, the chapter discusses generic real-time in more detail and includes an example of how to generate stand-alone C code from your Simulink block diagram.

The chapter continues with a description of what to do based on target type, including a brief discussion of some supported hardware. The chapter concludes with some information about the dSPACE target.

## Basic Concepts in the Real-Time Workshop

Before going through a step-by-step example of how to target specific hardware using the Real-Time Workshop, it is important to understand basic concepts involved in real-time applications. These include:

- Generic real-time
- Targeting
- Target files
- The Build process
- Template makefiles
- Configuring the template makefile
- Specifying model parameters
- Data logging
- Code validation

### Generic Real-Time

The Real-Time Workshop provides a generic real-time development system. Generic real-time is

- An environment for simulating fixed-step models in single or multi-tasking mode on your workstation
- A means by which you can perform code validation
- A starting point for targeting "in-house" or custom hardware

### Targeting

To use the Real-Time Workshop, you must decide on what type of environment you want to place the generated code. This is known as *targeting*. The *host* is where you run MATLAB, Simulink, and the Real-Time Workshop. Using the build tools on the host, you create code and an executable that runs on your target system, which is the computer system on which you execute your real-time application.

The system target file and template makefile define what type of target you have. The table below lists examples of target environments and the system target files and template makefiles associated with each example environment:

**Table 1-1: Target Systems and Associated Support Files**

| Target | Compiler | System Target File | Template Makefile |
|---|---|---|---|
| generic real-time executable on PC | Microsoft Visual C/C++ | `grt.tlc` | `grt_vc.tmf` |
| generic real-time project makefile | Microsoft Visual C/C++ | `grt.tlc` | `grt_msvc.tmf` |
| generic real-time executable on PC | Watcom C | `grt.tlc` | `grt_watc.tmf` |
| generic real-time on UNIX | Any ANSI C compiler supported on your UNIX system | `grt.tlc` | `grt_unix.tmf` |

## Target Files

*Target files*, which are files that are compiled and executed by the Target Language Compiler, describe how to create code for your target. The system target file is the entry point for the TLC program that creates the executable. The block target files define how the code looks for each of the Simulink blocks in your model.

## The Build Process

The Real-Time Workshop build process is controlled by `make_rtw`, which is invoked when you click on the **Build** button of the RTW page of the **Simulation parameters** dialog box. First, `make_rtw` compiles the block diagram and generates a `model.rtw` file. Next, `make_rtw` invokes the Target Language Compiler to generate the code. You must specify the system target file on the RTW page for the TLC. Then `make_rtw` creates a makefile, `model.mk`, from the template makefile specified in the RTW page. Finally, if the host on which you are running matches the `HOST` macro specified in the template makefile, `make` is invoked to create a program from the generated code. See

Chapter 3, "Code Generation and the Build Process," for more information on the build process.

## Template Makefiles

The Real-Time Workshop uses template makefiles to build an executable from the generated code. By convention, a template makefile has an extension of `.tmf` and a name corresponding to your target. For example `grt_unix.tmf` is the generic real-time template makefile for UNIX.

A makefile is created from the template makefile by copying each line from the template makefile, expanding tokens into the makefile. See "Template Makefile" on page 3-9 for more information about template makefiles and expandable tokens. The name of the makefile created is *model*`.mk`. The *model*`.mk` file is then passed to a `make` utility, which is a utility that builds an executable from a set of files. The `make` utility performs date checking on the dependencies between the object and C files and rebuilds the object files if needed.

## Template Makefile Configuration

You can configure the build process by modifying the template makefile. You can do this by copying it to your local working directory and editing it. Alternatively, you can configure the template makefile's operation, by specifying build options to the `make_rtw` command.

For example, in the RTW page of the **Simulation parameters** dialog box, you can turn debugging symbols on for the `grt_unix.tmf` by specifying the build command as:

```
make_rtw OPT_OPTS=–g
```

The comment section of the template makefiles distributed by The MathWorks includes some helpful hints about common build options.

## Specifying Model Parameters

You modify the model parameters that control aspects of the simulation, such as start and stop time, by altering the fields of the **Simulation Parameters** dialog box. The simulation parameters are used directly for code generation and program building. Therefore, before you generate code and build a program, you must verify that the model parameters have been set correctly in your simulation parameter dialog box.

- Verify your start and stop times are correct.
- You must be using a fixed-step solver.
- If you are creating a generic real-time executable, verify that you have selected the correct workspace I/O settings and that you have the appropriate Scope and/or To Workspace blocks.

## Data Logging

The generic real-time programs perform the same data logging as a Simulink simulation. When the generic real-time program is complete, a `model.mat` file is created containing all workspace variables that would have been created by running a simulation. The names of these workspace variables are equal to the names that would have been created by the simulation with a "`rt_`" prefixed to them. Aside from MAT-file data logging, you can also use *signal monitoring*. See "Data Logging and Signal Monitoring" on page 4-4 for a complete discussion.

## Code Validation

Using the generic real-time environment provided by The MathWorks, you can create an executable that runs on your workstation and creates a `model.mat` file. You can then compare the results of the generic real-time simulation with the results of running a Simulink simulation.

## Building Generic Real-Time Programs

This example describes how to generate C code from a Simulink model and create a generic real-time program. This program:

- Executes as a stand-alone program, independent of external timing and events.
- Saves data in a MATLAB MAT-file for later analysis.
- Is built in the UNIX and PC environments.

It also demonstrates how to use data logging to validate the generated code. Data logging allows you to compare system outputs from the Simulink block diagram to the data stored by the program created from the generated code.

### The Simulink Model

This example uses the `f14` Simulink model from the `simdemos` directory (note that, by default, this directory is on your MATLAB path). `f14` is a model of a flight controller for the longitudinal motion of a Grumman Aerospace F-14 aircraft.

**Naming Conventions:** This example is based on a model of the F-14 aircraft. When typed at the MATLAB prompt, `f14` opens the model. When passed to the operating system, `!f14` executes the file named `f14`. When used with the MATLAB load command, `load f14` reads from the file `f14.mat`. Finally, the phrase F-14 refers to the aircraft.

To display the model with Simulink, at the MATLAB command line, enter:

```
f14
```

This is the Simulink model that appears:

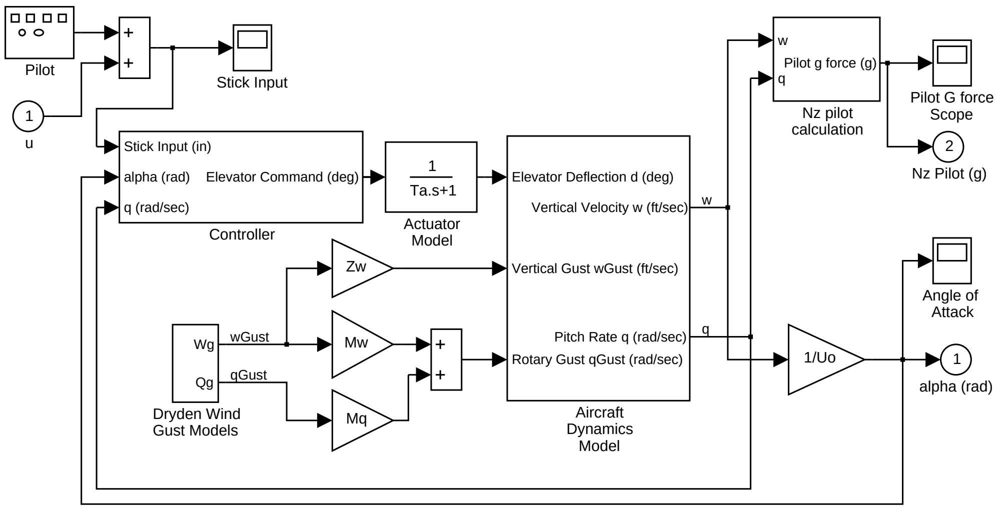


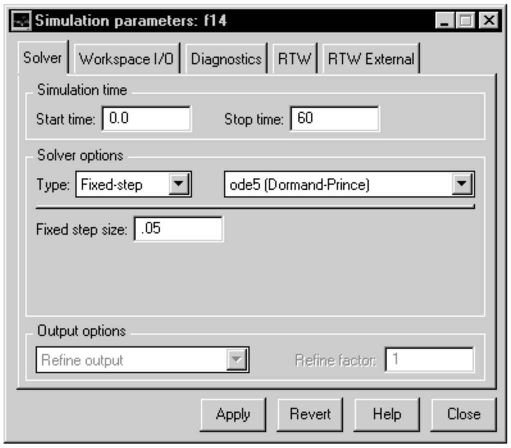
**Figure 2-1: Simulink Model of an F14 Aircraft**

The model employs a Signal Generator block to simulate the pilot's stick input, which is monitored by a Scope block. The system outputs are the aircraft angle of attack and the G forces experienced by the pilot. The outputs are also monitored by Scope blocks. These Scope blocks provide a means to monitor the operation of the Simulink model.

The design and internal workings of the `f14` model are not discussed in this example; such a discussion is beyond the scope of this manual. However, the procedure you must follow to use the Real-Time Workshop is generally independent of model content and complexity.

## Setting Program Parameters

After displaying the `f14` model, select **Parameters** from the **Simulation** pull-down menu. This displays the **Simulation parameters** dialog box with the Solver page selected by default:



The **Simulation parameters** dialog box allows you to set options, select a template makefile, generate code, and build the program.

When initially displayed, the **Simulation parameters** dialog box contains default values that do not necessarily match the simulation parameters required to create a generic real-time executable. You should, therefore, change the default parameters to match the simulation parameters that are needed to use your model with the Real-Time Workshop. To configure the options for the `f14` model, change the following parameters:

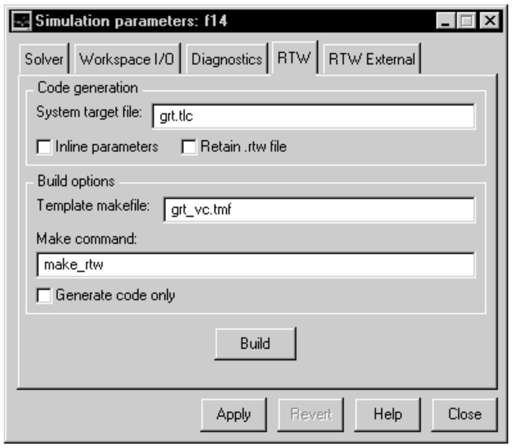
1 Under the Solver tab, set the **Solver options Type** to `Fixed-step` and select the **ode5 (Dormand-Prince)** solver.

2 Set the **Fixed Step Size** to `0.05`

**Note:** Alternatively, you can select **RTW Options** from the **Tools** pull-down menu. This brings you directly to the RTW page of the **Simulation parameters** dialog box.

## Building the Program

To build the program, select the RTW page in the **Simulation parameters** dialog box:


Simulation parameters: f14

Solver | Workspace I/O | Diagnostics | RTW | RTW External

Code generation
System target file: grt.tlc
☐ Inline parameters ☐ Retain .rtw file

Build options
Template makefile: grt_vc.tmf
Make command: make_rtw
☐ Generate code only

Click the **Build** button to generate C code from the `f14` model. The build command calls a system target file, `grt.tlc`, that uses the specified template (by default `grt_vc.tmf` on PC and `grt_unix.tmf` on UNIX platforms) to create the makefile, which is then used to build the program.

**Note:** Alternatively, you can select **RTW Build** from the **Tools** pull-down menu. This directs the Real-Time Workshop to generate code for the current model and is equivalent to clicking the **Build** button on the RTW page.

When you click on the **Build** button in the **Simulation parameters** dialog box, the Real-Time Workshop invokes the `make` command, which in turn:

- Compiles the block diagram to produce the `model.rtw` file (in this example, `f14.rtw`)
- Invokes the Target Language Compiler, which in turn compiles the TLC program, starting with `grt.tlc`, and operates on *`model`*`.rtw` to produce the generated code.
- Creates a makefile called *`model`*`.mk` (e.g., `f14.mk`) from the template makefile (e.g., `grt_vc.tmf` or `grt_unix.tmf`)
- If Simulink is running on the same type of host as that specified in the template makefile, then the program is built. Otherwise, processing stops after creating the model code and the makefile.

Once you have executed the **Build** command, the Real-Time Workshop creates these files by default:

- `f14.c` – the stand-alone C code
- `f14.h` – an include header file containing information about the state variables
- `f14.reg` – an include file that contains the model registration function responsible for initializing data structures in the generated code
- `f14.prm` – an include file that holds information about the parameters used in the f14 model
- `f14` on UNIX and `f14.exe` on PC – the generic real-time executable

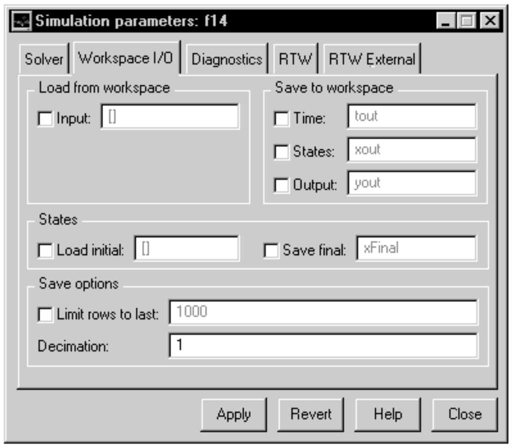
## Customizing the Build Process

You can choose to inline parameters by clicking the **Inline parameters** check box on the RTW page. This directs the Real-Time Workshop to substitute the numerical values of the model parameters in place of the variable names. It also instructs Simulink to propagate constant sample times, which improves performance by placing constant operations into the start-up code. You can also customize the template makefile and the `make` commands; these options are discussed later in this chapter.

Clicking **Generate code only** tells the Real-Time Workshop to generate the C code without compiling, meaning that object and executable files are not created. Clicking **Retain .rtw file** directs the Real-Time Workshop to save the *model*`.rtw` file. By default, this file is deleted during the build process. For more information about customizing the build process, see “Automatic Program Building” on page 3-3.

### Data Logging

This example uses the Real-Time Workshop’s MAT-file Data Logging facility to save system states and outputs, as well as the simulation time. To do this, select the Workspace I/O page of the Simulation Parameters dialog box. Checking the Time, States, and Outputs options directs the Real-Time Workshop to log the simulation time, any discrete or continuous states, and any root Outport blocks. This is a picture of the Workspace I/O page:



Load from workspace
☐ Input: []

Save to workspace
☐ Time: tout
☐ States: xout
☐ Output: yout

States
☐ Load initial: []
☐ Save final: xFinal

Save options
☐ Limit rows to last: 1000
Decimation: 1

For example, if you checked the Time and Outputs logging options, the Real-Time Workshop saves the inputs to all Scope blocks that are configured to save data to the workspace, the simulation time, and the root Output blocks in a MATLAB MAT-file. You can load this data into MATLAB for analysis. See “Code Validation” on page 2-16 for more information.

Inputs to models cannot be saved using the Workspace I/O page. To save inputs, you can use Scope block and click the **Properties** button. Alternatively, you can use a To Workspace block. See *Using Simulink* for more information about saving data to the MATLAB workspace using either of these methods.

## Run-time Interface for Real-Time

Building the real-time program requires a number of source files in addition to the generated code. These source files contain:

- A main program
- Code to drive execution of the model code
- Code to implement an integration algorithm
- Code to carry out data logging
- Code to create the `SimStruct` data structure, which is used to manage execution of your model

## Configuring the Template Makefile

This example uses two different template makefiles: `grt_unix.tmf` for the UNIX environment and `grt_vc.tmf` for the PC environment. These makefile templates are automatically processed into makefiles properly configured to build the program. However, makefiles generally require some degree of configuration before you use them.

These files are located in the `matlab/rtw/c/grt` directory. Use the MATLAB `matlabroot` command to determine the MATLAB path on your system.

### The UNIX Template Makefile

The `grt_unix.tmf` template specifies `cc` or GNU's `gcc` as the default compiler. It also specifies the necessary source files, compiler flag, include paths, etc., that are required to build the program. You can, in general, use this makefile template without modification.

`grt_unix.tmf` is designed to be used by GNU Make, which is located in *matlabroot*`/bin/`*arch* (except for Linux systems, which provide GNU Make by default).

### The PC Template Makefiles

The template makefiles `grt_vc.tmf` and `grt_msvc.tmf` are designed to be used with the Microsoft Visual C/C++ compiler. To use either of these template makefiles, you must verify that the Visual C/C++ environmental variables are defined and that nmake is on your search path. To use grt_vc.tmf or grt_msvc.tmf, you must verify:

1 That the MsDevDir environmental variable is defined

2 That the Visual C/C++ nmake utility is on your path

The template makefile grt_watc.tmf is designed to be used with the Watcom C/C++ Compiler and the Watcom make utility, wmake. To use grt_watc.tmf, you must verify:

1 That the WATCOM environment variable is defined.

2 That the Watcom wmake utility is on your search path before you build the program.

## Generic Real-Time Modules

These source modules are automatically linked by the makefile, `model.mk`. The modules used to build this example are:

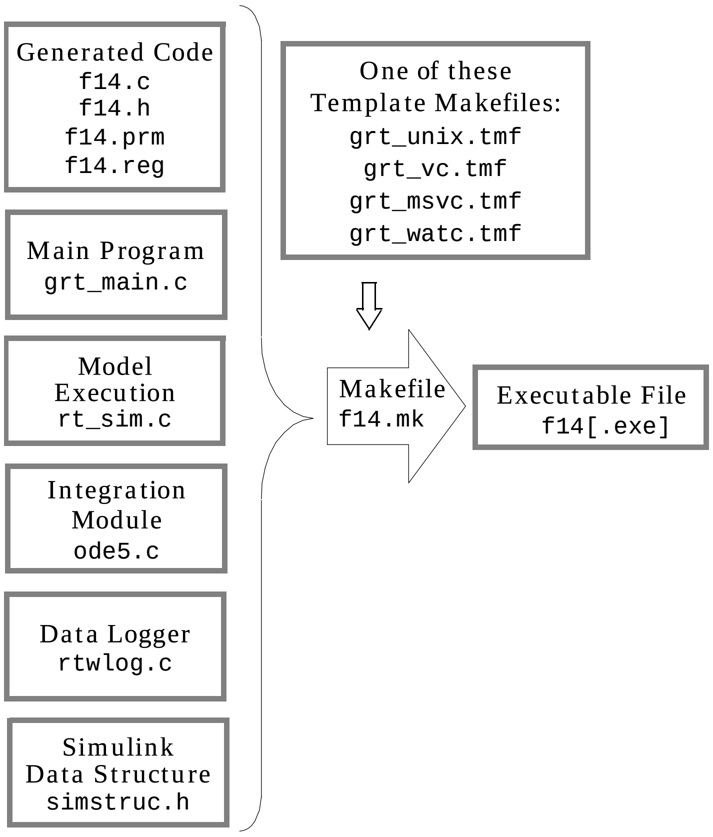


**Figure 2-2: Source Modules Used to Build the Program**

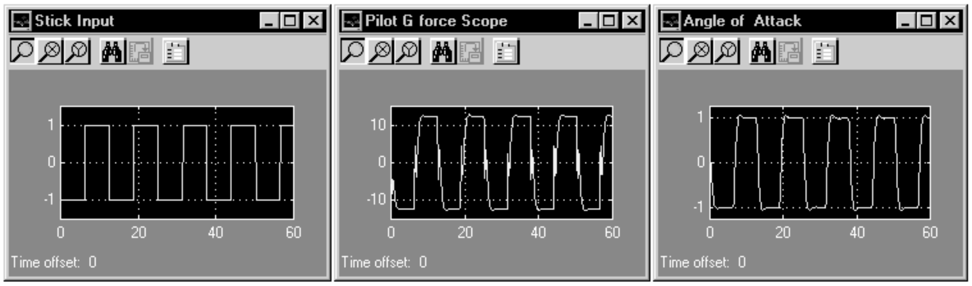
This diagram illustrates the code modules that are used to build a generic real-time program.

## Code Validation

After completing the build process, the stand-alone version of the `f14` model is ready for comparison with the Simulink model. The MAT-file data logging options selected with the Workspace I/O page of the **Simulation parameters** dialog box cause the program to save the pilot G forces, aircraft angle of attack, and simulation time. You can save the Stick Input signal to the MATLAB workspace by setting options in the Scope block. You can now use MATLAB to produce plots of the same data that you see on the three Simulink scopes.

In both the Simulink and the generic real-time stand-alone executable version of the `f14` model, the stick input is simulated with a square wave having a frequency of 0.5 (rad/sec) and an amplitude of plus and minus 1, centered around zero.

Opening the Stick Input, Pilot G Force, and Angle of Attack scopes and running the Simulink simulation from `T = 0` to `T = 60` produces:



Now run the stand-alone program from MATLAB:

```
!f14
```

The "!" character passes the command that follows it to the operating system. This command, therefore, runs the stand-alone version of `f14` (not the M-file).

To obtain the data from the stand-alone program, load the file `f14.mat`:

```
clear
load f14
```

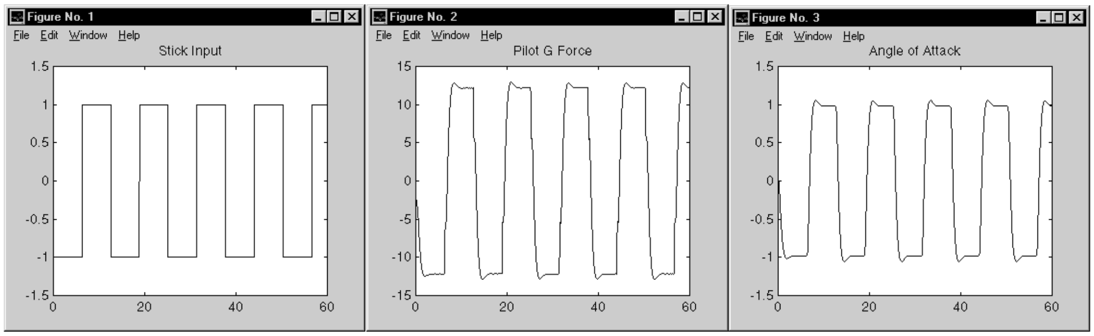
Then look at the workspace variables:

```
who
Your variables are:
rt_Pilot_G_force    rt_tout
rt_Angle_of_attack  rt_xout
rt_Stick_input      rt_yout
```

The variables `rt_tout`, `rt_xout`, and `rt_yout` were logged because the appropriate time, states, and outputs buttons were clicked on the Workspace I/O page. The variables `rt_Pilot_G_force`, `rt_Angle_of_attack`, and `rt_Stick_input` were saved using the Properties page of the Scope blocks in the `f14` model. If you are unfamiliar with how to save variables using Scope blocks, refer to the Simulink documentation. The variables are named according to the Simulink block that produced them, with spaces changed to underscores and an `rt_` prefix added to the variable names to identify them as real-time data.

You can now use MATLAB to plot the three workspace variables as a function of time:

```
plot(rt_tout,rt_Stick_input(:,2))
figure
plot(rt_tout,rt_Pilot_G_force(:,2))
figure
plot(rt_tout,rt_Angle_of_attack(:,2))
```



## Comparing Simulation and Generic Real-Time Results

Your Simulink simulations and generic real-time code should produce nearly identical output. Let's compare the `f14` model output from Simulink to the results achieved by the Real-Time Workshop. Follow these steps to do a correct comparison:

- First, make sure that you've selected the same (fixed-step) integration scheme for both the Simulink run and the Real-Time Workshop build process (for example, `ode5` (Dormand-Prince). Also, set the Fixed step size to the same number (for example, 0.05).
- Set the Scope blocks to log the input and outputs of the `f14` simulation (for example, you can set the Angle of Attack Scope block output to `Angle_of_attack`, the Pilot G Force Scope block output to `Pilot_G_Force`, and the Stick Input Scope block output to `Stick_input`).
- Run the `f14` simulation.
- Build the `f14` generic real-time system, run it, and load the MAT-file.

You should now have two sets of data, one generated by Simulink and one generated by the Real-Time Workshop. At the MATLAB prompt, type:

```
who
```

You should see among the variables in your workspace:

```
Angle_of_attack                 rt_Stick_input
Pilot_G_force                   Stick_input
rt_Angle_of_attack
rt_Pilot_G_force
```

Comparing `Angle_of_attack` to `rt_Angle_of_attack` produces:

```
max(abs(rt_Angle_of_attack-Angle_of_attack))
ans =
  1.0e-015 *
         0    0.5551
```

Comparing `Pilot_G_force` to `rt_Pilot_G_force` produces:

```
max(abs(rt_Pilot_G_force-Pilot_G_force))
ans =
  1.0e-012 *
         0    0.1007
```

So overall agreement is within $10^{-12}$. This slight error is caused by many factors, including

- Different compiler optimizations
- Statement orderings
- Run-time libraries

For example, sin(2.0) may differ depending on which C library you are using.

## Analyzing Data with MATLAB

For a more detailed analysis, you can add To Workspace blocks to the Simulink model. These blocks allow you to access any data generated within the block diagram and save it in MATLAB workspace variables. Alternatively, you can connect scope blocks configured to log data to the model.

## Targeting dSPACE

dSPACE, a company that specializes in real-time hardware and software products, markets a complete set of tools for use with the Real-Time Workshop. These tools are primarily intended for rapid prototyping of control systems and hardware-in-the-loop applications. This section provides a brief example of how to use dSPACE's RTI (real-time interface) with the Real-Time Workshop.

The dSPACE hardware line includes DSP boards based on the Texas Instruments TMS 320 C31 and C40 DSP's. dSPACE boards include:

- A single processor board, the DS1102, that includes on-board I/O (A/D and D/A conversion as well as digital I/O and encoder inputs).
- Multiprocessor boards, such as the quad C40 board.
- The DS1004 based on the DEC Alpha, used for maximum throughput.

For a current summary and more information on dSPACE products and their relationship to the Real-Time Workshop, refer to the documentation provided by dSPACE or contact your dSPACE distributor.

At present, all dSPACE boards conform to the half length or full length ISA slot size. The ISA slot enables the host PC to download the executable to a dSPACE target processor board. It provides higher bandwidth than is possible over the ISA backplane. dSPACE boards utilize the proprietary PHS bus; this bus lets the dSPACE processor board(s) communicate with other dSPACE I/O boards. Multiprocessor and multi-I/O boards are common. One exception is that the DS1102 is for use as a single board solution with its own on-board I/O. The ISA bus is also used for downloading new block diagram parameters and for passing collected data back to the host computer.

The dSPACE product line includes a variety of A/D and D/A boards, encoder boards, and digital I/O boards. A hardware prototyping board gives you the ability to interface your own hardware directly with the dSPACE hardware. You can use all these products in combination with code generated by the Real-Time Workshop.

The following two pictures show a PC setup that includes target hardware in the PC chassis and the dSPACE PHS bus:

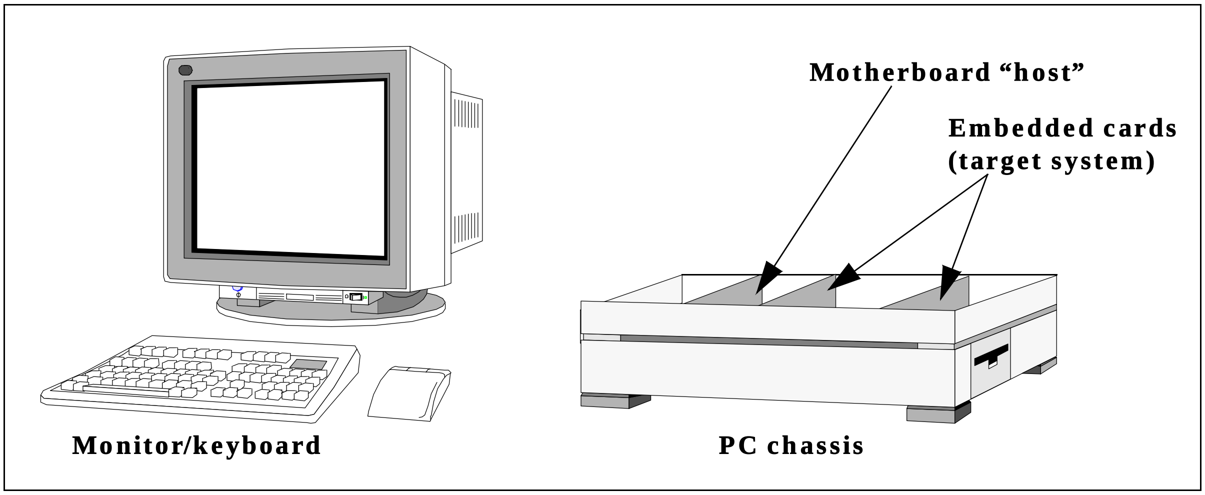


**Figure 2-3: A PC Setup Including a Target System**

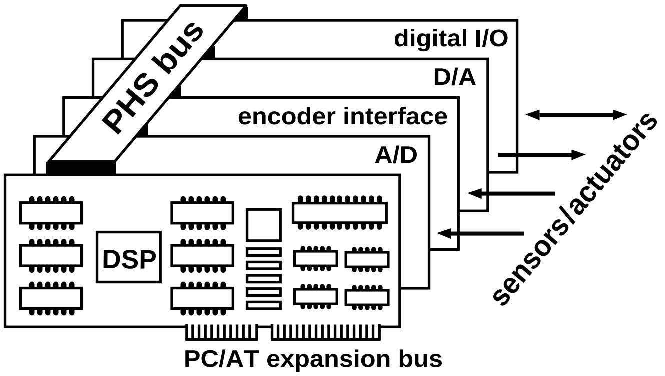


**Figure 2-4: The dSPACE PHS Bus**

### Real-Time Interface (RTI)

When using the Real-Time Workshop with dSPACE hardware, you must develop target-specific software. The key software component for this configuration is the dSPACE Real-Time Interface to Simulink. The RTI, with special versions available for various dSPACE processor boards (C31, C40, DEC Alpha, etc.), is available directly from dSPACE and dSPACE distributors.

Executing code generated from the Real-Time Workshop on a particular target in real-time requires target-specific code. Target-specific code includes I/O device drivers and an interrupt service routine. Other components, such as a communication link with Simulink, are required if you need the ability to download parameters on-the-fly to your target hardware. The dSPACE RTI provides this support. Since these components are specific to particular hardware targets (in this case dSPACE hardware), you must ensure that these target-specific components are compatible with the target hardware. To allow you to build an executable, dSPACE also provides a target makefile specific to a particular dSPACE processor board. This target makefile invokes the crosscompiler, which is also available from dSPACE.

When used in combination with RTW, dSPACE products provide an integrated environment that, once installed, needs no additional coding.

### Building a Real-Time Executable with dSPACE's RTI

dSPACE provides a set of preconfigured files for use with their hardware and RTI. These files simplify the process of building a real-time executable and downloading it to the dSPACE target microprocessor. For operation with dSPACE, refer to the dSPACE documentation.

From the MATLAB command line, type `dslib`. This opens a Simulink block diagram that includes a set of preconfigured blocks for dSPACE I/O device drivers. These devices are associated with particular dSPACE hardware. Add the I/O device drivers as appropriate for your model and for your dSPACE hardware.

Select the pull-down menu for the **Simulink parameters** dialog box. From this dialog box, select the RTW page. In the following example, it is assumed that you are using the dSPACE DS1102 C31 board. When using other dSPACE processor boards, you must specify the appropriate versions of the system target file and template makefile. For the current example, in the RTW page of the dialog box, specify:

- System target file: `rt31.tlc`
- Template makefile: `rti31.tmf`

With this configuration, you can now generate a real-time executable and download it to the dSPACE processor board. You can do this by clicking the **Build** button on the RTW page. The Real-Time Workshop automatically generates C code and inserts the I/O device drivers as specified in your block diagram. These device drivers are inserted in the generated C code as inline S-functions. Inlined S-functions offer advantages in speed and simplify the generated code (for more information about inlining S-functions, refer to the *Target Language Compiler Reference Guide*).

During the same build operation, the template makefile and dialog entries are combined to form a target makefile for your dSPACE setup. This makefile invokes the TI cross-compiler and builds an executable that is automatically downloaded via the ISA bus onto the dSPACE processor board. With additional support tools from dSPACE, such as TRACE and COCKPIT, you can view signals, collect data, and log data in a MATLAB MAT-file format.

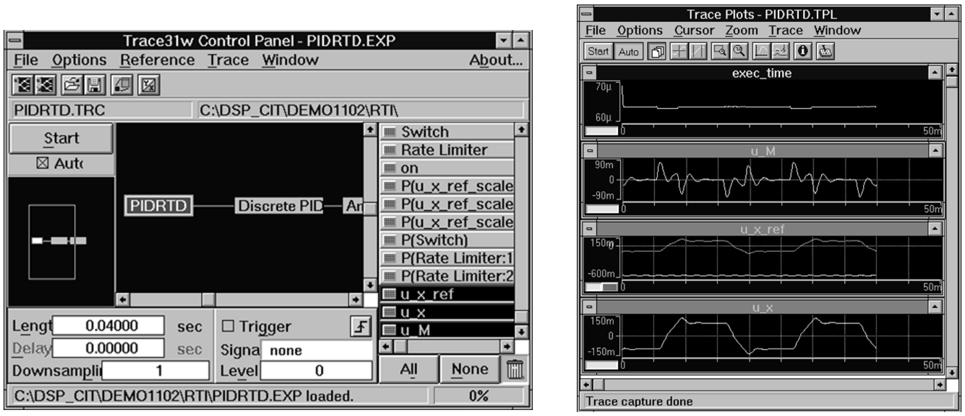
You can also change model parameters while the model runs on the target processor. There are two ways to do this:

- Using Simulink's external mode
- Using dSPACE'S COCKPIT software

### dSPACE TRACE

TRACE enables you to capture and view signals using names that appear in the Simulink block diagram. Multiple signals can be acquired and displayed from a convenient GUI. TRACE also includes a signal browser that helps users scan the model hierarchy and that selects signals to monitor from a particular subsystem. Options such as data decimation and data acquisition triggering are available from TRACE. Once data is collected, you can save it in a MATLAB MAT-file format for easy import into MATLAB.

This picture shows the TRACE Control Panel and a sample of TRACE plots:



PIDRTD.TRC C:\DSP_CIT\DEMO1102\RTI\

Length 0.04000 sec
Delay 0.00000 sec
Downsampling 1
Signal none
Level 0

C:\DSP_CIT\DEMO1102\RTI\PIDRTD.EXP loaded. 0%

Trace capture done

**Figure 2-5: The TRACE Control Panel and Sample TRACE Plots.**

### dSPACE COCKPIT

COCKPIT provides a virtual instrument panel on the host PC. It provides an interface for editing parameters while the generated code executes on the dSPACE processor boards. The communications link between COCKPIT and the processor boards lets you display signals with dials and alert displays or change parameters using pushbuttons, sliders, and other input blocks on the COCKPIT instrument panel.

3

# Code Generation and the Build Process

## Introduction

The Real-Time Workshop simplifies the process of building application programs. One of the Real-Time Workshop's features is *automatic program building*, which provides a standard means to create programs for real-time applications in a variety of host environments. It does this in a uniform and controlled manner, yet is customizable to different applications.

Automatic program building uses the `make` utility to control how the program is built and an M-file to create a customized makefile (the description file referenced by the `make` utility) from a customizable template.

Chapter 2, "Getting Started with the Real-Time Workshop," introduced the build process and template makefiles. This chapter discusses these concepts in more detail. Topics included are:

- Automatic program building
- The Real-Time Workshop user interface
- Configuring generated code
- Customizing template makefiles
- Generic real-time template makefiles

## Automatic Program Building

The Real-Time Workshop automates the task of building a stand-alone program from your Simulink model. When you click on the **Build** button on the RTW Page of the **Simulation parameters** dialog box, the `make` command, which starts with `make_rtw`, is invoked. The build process consists of three main steps that are controlled by an M-file, `make_rtw.m`:

1. Generating the model code.

2. Generating a makefile that is customized for a given build.

3. Invoking the `make` utility with the customized makefile.

The shaded box in the figure below outlines these steps:

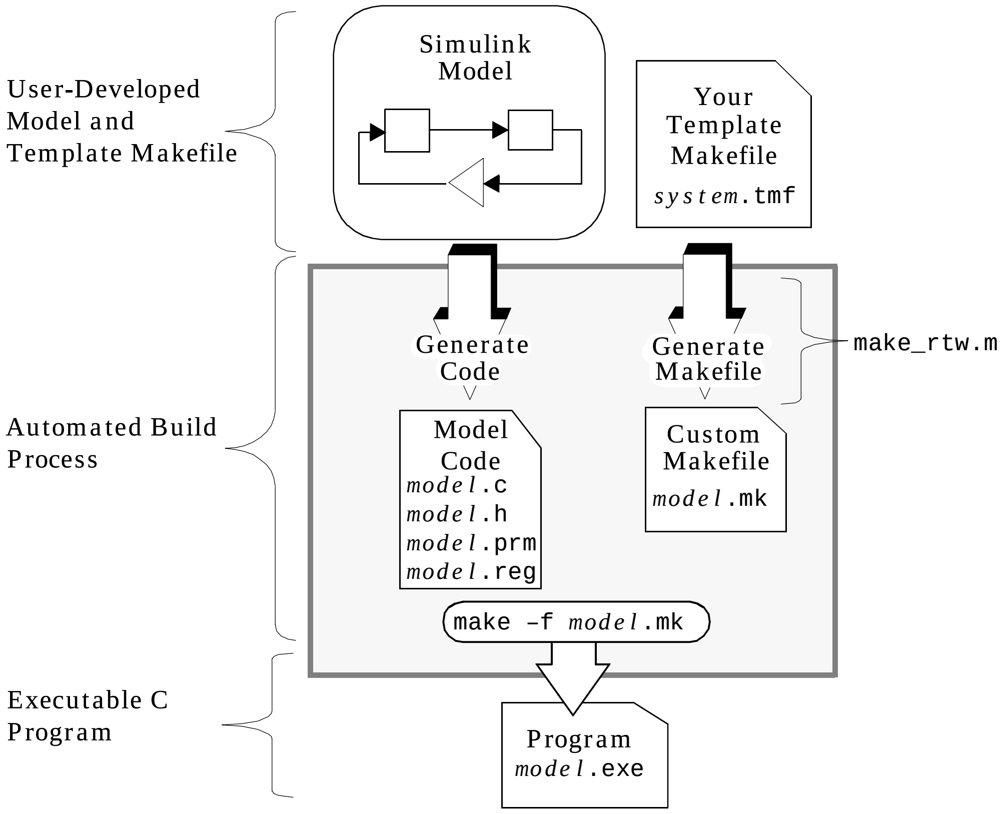


**Figure 3-1: The Build Process**

The task of creating a stand-alone program from the generated code is simplified by automated program building.

When you click the *Build* button on the RTW page on the **Systems parameters** dialog box, these steps are carried out automatically. This diagram illustrates the logic the controls this process:

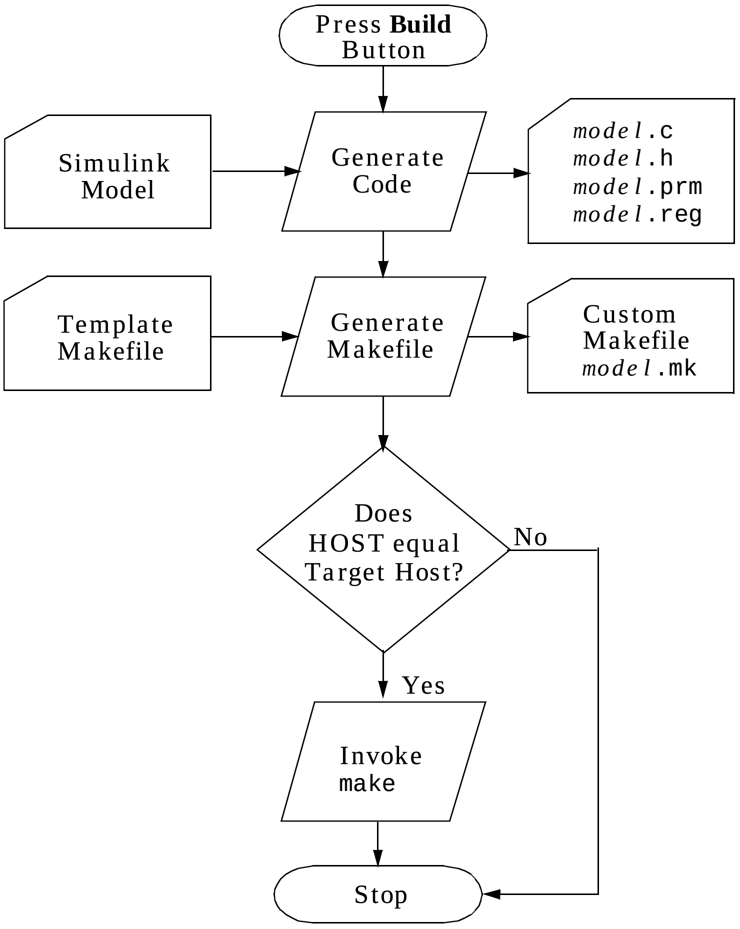


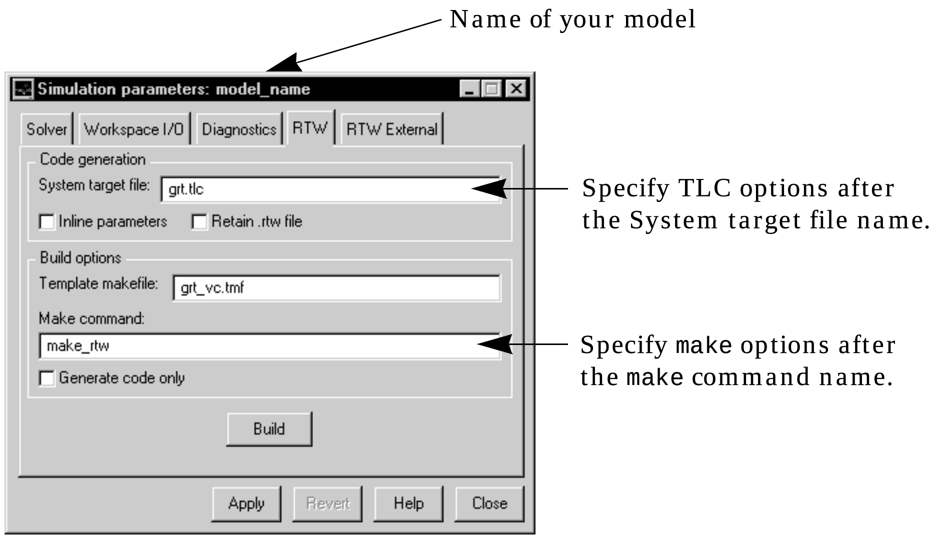
**Figure 3-2: The Logic That Controls Automatic Program Building**

## The Real-Time Workshop User Interface

You work with the Real-Time Workshop by interacting with and modifying fields of the **Simulation parameter** dialog box of your Simulink model. Two pages of this dialog box are exclusively for the Real-Time Workshop. The other pages apply to both Simulink simulations and the Real-Time Workshop. To access the **Simulation parameters** dialog box, you can select the Parameters item of the **Simulation** menu. Alternatively, you can select the RTW item from the **Tools** menu, which opens the simulation parameters dialog box to the RTW page.

All pages of the **Simulation parameters** dialog box affect RTW code generation. On the Solver page, you specify the type of solver, and its options including start and stop time. When using the Real-Time Workshop you must specify a fixed-step solver. To specify data logging options, you use the Workspace I/O page (providing you target has support for these options). On the Diagnostics page you specify options that affect whether or not various model conditions such as unconnected ports are ignored, treated as a warning, or cause an error condition when the build process is invoked.

This chapter discusses in detail the RTW page of the **Simulation parameters** dialog box, including the fields on the page and their optional arguments. The RTW External page is for use with external mode, which is described in Chapter 4, “External Mode, Data Logging, and Signal Monitoring,”


Name of your model
Simulation parameters: model_name
Solver
Workspace I/O
Diagnostics
RTW
RTW External
Code generation
Build options
Build
Apply
Revert
Help
Close
Specify TLC options after the System target file name.
Specify make options after the make command name.

System target file: grt.tlc
☐ Inline parameters ☐ Retain .rtw file
Template makefile: grt_vc.tmf
Make command: make_rtw
☐ Generate code only

**Figure 3-3: Description of the RTW Page**

### System Target File

Use the System Target File field to specify what type of code and target for which you are generating code. For example,

- `grt.tlc` is the system target file for the *generic real-time* target used for generating code targeted for your workstation.
- `tornado.tlc` is the system target file for the *Tornado* (VxWorks) real-time target.
- `drt.tlc` is the system target file for the *DOS* real-time target.

After the system target file, you can specify options for the Target Language Compiler (TLC). The common options are:

**Table 2-1: Target Language Compiler Options**

| Option | Description |
|---|---|
| `–I`*`path`* | Adds *path* to the list of paths in which to search for target files (`.tlc` files). |
| `–m[`*`N`*`\|a]` | Maximum number of errors to report when an error is encountered (default is 5). For example, `-m3` specifies that at most three errors will be reported. To report all errors, specify `–ma`. |
| `–d[g\|n\|o]` | Specifies debug mode (generate, normal, or off). Default is off. When `–dg` is specified, a `.log` file is create for each of your TLC files. When debug mode is enabled (i.e., generate or normal), the Target Language Compiler displays the number of times each line in a target file is encountered. |
| `–a`*`Variable`*`=`*`expr`* | Assigns an variable to a specified value (i.e., creates a parameter value pair) for use by the target files during compilation. |

If your system target file is using block target files and TLC library provided by The MathWorks, there are several TLC variables that can be defined that alter the generated code. The variables are:

**Table 2-2: Target Language Compiler Optional Variables**

| Variable | Description |
|---|---|
| `-aRollThreshold=`*N* | Specifies the threshold for the `%roll` TLC operator. The `%roll` is used to selectively inline or place a sequence of statements in a "for" or "do" loop. If the number of statements is equal to or exceeds the threshold, then the statements are placed in a for loop. |
| `-aFileSizeThreshold=`*N* | Specifies a threshold which when exceeded will cause the *model*`.c` file to be split into a *model1*`.c`. Likewise, *model1*`.c` will be split into *model2*`.c` if the threshold is exceeded, and so on. |

## Inline Parameters

*Inlining parameters* refers to a mode where blocks with a constant sample time are removed from the run-time model execution. The output signals of these blocks are set up once during model start up. When you select this option, you should also turn "invariant constants" on when simulating your model within Simulink. See the Simulink documentation for information on invariant constants.

## Retaining the model.rtw File

When modifying the target files, you will need to look at the *model*`.rtw` file. To prevent the *model*`.rtw` file from being deleted after the build process is completed, select the **Retain .rtw file** check box.

## Template Makefile

The template makefile is used when the **Generate code only** check box is not selected. The template makefile uniquely identifies which target for which you are creating an executable.

### Make Command

The make command starts with make_rtw (unless this has been replaced with a third-party make command, in which case you should see their documentation). This command is invoked when you click the **Build** button. You can supply make arguments as additional arguments to make_rtw. For example, if you are using one of the grt*.tmf files, you can alter the optimization options by specifying:

```
make_rtw OPT_OPTS="compiler_specific_setting"
```

## Configuring the Generated Code

You can configure the generated code by appending TLC options to the system target filename in the System Target file field on the RTW page of the **Simulation parameters** dialog box (see “System Target File” on page 3-7). Your target configuration may have additional options. Options which can be altered are generally described in your system target file. If you are using a system target file provided by a 3rd party vendor, please consult their documentation.

In addition to TLC options, you can customize all aspects of the generated code by modifying the target (`.tlc`) files. These files are located in `matlabroot/rtw/c/tlc`. Modification of the target files is described in detail in the *Target Language Compiler Reference Guide*.

## Template Makefiles

This section contains a description of how to work with and modify the template makefiles that are used with the Real-Time Workshop. Template makefiles are essentially makefiles in which certain tokens are expanded to create a makefile for your model (*model*`.mk`). *model*`.mk` is created from the template makefile specified in the RTW page of the **Simulation parameters** dialog box by copying each line from the template makefile and expanding the tokens of Table 3-1.

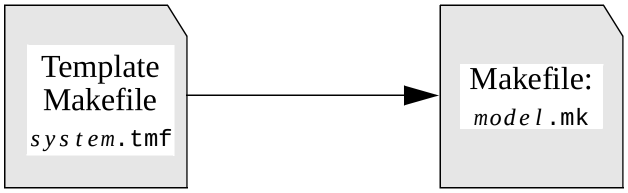


**Figure 3-4: Creation of model.mk**

*model*`.mk` is created from your target systems template makefile by copying line for line the contents of it and expanding the RTW tokens. This table lists the RTW tokens and their expansions:

**Table 3-1: Template Makefile Tokens Expanded by make_rtw**

| Token | Expansion |
|---|---|
| `\|>MODEL_NAME<\|` | Name of the Simulink block diagram currently being built. |
| `\|>MODEL_MODULES<\|` | Any additional generated source (`.c`) modules. For example, you can split a large model into two files, *model*`.c` and *model*`1.c`. In this case, this token expands to *model*`1.c`. |
| `\|>MODEL_MODULES_OBJ<\|` | Object filenames (`.obj`) corresponding to any additional generated source (`.c`) modules. |

**Table 3-1: Template Makefile Tokens Expanded by make_rtw (Continued)**

| Token | Expansion |
|---|---|
| `\|>MAKEFILE_NAME<\|` | *model*`.mk` - The name of the makefile that was created from the template makefile. |
| `\|>MATLAB_ROOT<\|` | Path to where MATLAB is installed. |
| `\|>MATLAB_BIN<\|` | Location of the MATLAB executable. |
| `\|>S_FUNCTIONS<\|` | List of noninlined S-function (`.c`) sources. |
| `\|>S_FUNCTIONS_OBJ<\|` | Object (`.obj`) file list corresponding to noninlined S-function sources. |
| `\|>SOLVER<\|` | Solver source filename, e.g., `ode3.c`. |
| `\|>SOLVER_OBJ<\|` | Solver object (`.obj`) filename, e.g., `ode3.obj`. |
| `\|>NUMST<\|` | Number of sample times in the model. |
| `\|>TID01EQ<\|` | Either 1 or 0: Are the sampling rates of the continuous task and the first discrete task equal? |
| `\|>NCSTATES<\|` | Number of continuous states. |
| `\|>BUILDARGS<\|` | Options passed to `make_rtw`. This token is provided so that the contents of your *model*`.mk` file will change when you change the build arguments, thus forcing an update of all modules when your build options change. |
| `\|>COMPUTER<\|` | Computer type. See the MATLAB `computer` command. |

After creating *model.mk* from your template makefile, the Real-T ime Workshop invokes a `make` command. Make is a utility designed to create an executable from a set of source files. To invoke make, the Real-Time Workshop issues this command:

```
makecommand -f model.mk
```

*makecommand* is defined by the MAKE macro in your systems template makefile (see Figure 3-4 on page 3-17). You can specify additional options to make as described in the “Make Command” on page 3-10. For example, specifying OPT_OPTS=-O2 to make_rtw generates the following make command:

```
makecommand -f model.mk OPT_OPTS=-O2
```

Typically, build options are specified as a comment at the top of the template makefile you are using.

You need to configure your template makefile if the options that can be passed to make do not provide you with enough flexibility. The Real-Time Workshop uses make since it is a very flexible tool that lets you control nearly every aspect of building the generated code and run-time interface modules into your real-time program.

### Make Utilities

To configure your template makefile, you need to understand how make works and how make processes makefiles. There are several good books on make. Consult your local book store or refer to the documentation provided with the make utility you are using.

There are several different versions of make available. Perhaps the most flexible and powerful make utility is GNU Make, which is provided by the Free Software Foundation. We provide GNU Make for both UNIX and PC platforms; it can be found in:

```
matlabroot/rtw/bin/arch
```

It is possible to use other versions of make with the Real-Time Workshop, but they do not have as rich a set of features. To work with the Real-Time Workshop, any version of make must allow this command format:

```
makecommand -f model.mk
```

### Structure of the Template Makefiles

Before configuring or creating a Real-Time Workshop template makefile, you should become familiar with its structure. A template makefile has four sections:

- The first section contains an initial comment section that describes what this makefile targets.
- The second section defines macros that tells `make_rtw` how to process the template makefile. The macros are:

  `MAKE` — This is the command used to invoke the make utility. For example, if

  ```
  MAKE = mymake
  ```

  then the `make` command that will be invoked is:

  ```
  mymake -f model.mk
  ```

  `HOST` — What platform this template makefile is targeted for. This can be `HOST=PC`, `UNIX`, `computer_name` (see the MATLAB `computer` command), or `ANY`.

  `BUILD` — This tells `make_rtw` whether or not (`BUILD=yes` or `no`) it should invoke `make` from the Real-Time Workshop build procedure.

  `SYS_TARGET_FILE` — Name of the system target file. This is used for consistency checking by `make_rtw` to verify that the correct system target file was specified in the RTW page of the **Simulation Parameters** dialog box.

  `BUILD_SUCCESS` — an optional macro that you can use to specify the build success string to be used when looking for successful completion on the PC. For example:

  ```
  BUILD_SUCCESS = ### Success creation of
  ```

  `BUILD_ERROR` — an optional macro that you can use to specify the build error message to be displayed when an error is encountered during the `make` procedure. For example,

  ```
  BUILD_ERROR = ['Error while building ', modelName]
  ```

  `DOWNLOAD` - an optional macro that you can specify as yes or no. If specified as yes (and `BUILD=yes`), then `make` is invoked a second time with the download target:

  ```
  make -f model.mk download
  ```

DOWNLOAD_SUCCESS — an optional macro that you can use to specify the download success string to be used when looking for a successful download. For example:

```
DOWNLOAD_SUCCESS = ### Downloaded
```

DOWNLOAD_ERROR — an optional macro that you can use to specify the download error message to be displayed when an error is encountered during the download. For example,

```
DOWNLOAD_ERROR = ['Error while downloading ', modelName]
```

- The third section defines the tokens make_rtw expands.
- The fourth section contains the make rules used in building an executable from the generated source code. The build rules are typically specific to your version of make.

The general structure of a template makefile is shown in Figure 3-4 on page 3-17:

```
#-- Section 1: Comments -----------------------------------------------------
#
# Description of what type of target and version of make this template makefile
# is for and any optional build arguments.
#


#-- Section 2: Macros read by make_rtw ---------------------------------------
#
# The following macros are read by the Real-Time Workshop build procedure:
#
#  MAKE            - This is the command used to invoke the make utility
#  HOST            - What platform this template makefile is targeted for
#                    (i.e., PC or UNIX)
#  BUILD           - Invoke make from the Real-Time Workshop build procedure
#                    (yes/no)?
#  SYS_TARGET_FILE - Name of system target file.

MAKE            = make
HOST            = UNIX
BUILD           = yes
SYS_TARGET_FILE = system.tlc
#-- Section 3: Tokens expanded by make_rtw -----------------------------------
#

MODEL           = |>MODEL_NAME<|
MODULES         = |>MODEL_MODULES<|
MAKEFILE        = |>MAKEFILE_NAME<|
MATLAB_ROOT     = |>MATLAB_ROOT<|
…
COMPUTER        = |>COMPUTER<|
BUILDARGS       = |>BUILDARGS<|

#-- Section 4: Build rules ---------------------------------------------------
#
# The build rules are specific for your target and version of make.
#
```

**Figure 3-5: Structure of a Template Makefile**

## Customizing and Creating Template Makefiles

To customize or create a new template makefile, you can copy an existing template makefile to your local working directory and modify it.

The `make` utility processes the *model*`.mk` makefile and generates a set of commands based upon dependencies defined in *model*`.mk`. For example, to build a program called `test`, `make` must link the object files. However, if the object files don't exist or are out of date, `make` must compile the C code. After `make` generates the set of commands needed to build or rebuild `test`, `make` executes them.

Each version of `make` differs slightly in its features and how rules are defined. For example, consider a program called `test` that gets created from two sources, `file1.c` and `file2.c`. Using most versions of `make` the dependency rules would be:

```
test: file1.o file2.o
        cc -o test file1.o file2.o

file1.o: file1.c
        cc -c file1.c

file2.o: file2.c
        cc -c file2.c
```

In this example, we assumed a UNIX environment. In a PC environment the file extensions and compile and link commands will be different. The first rule, `test: file1.o file2.o` encountered by `make` will be built. In processing this rule, `make` sees that to build `test`, it needs to build `file1.o` and `file2.o`. To build `file1.o`, `make` processes the rule `file1.o: file1.c` and will compile `file1.c` if `file1.o` doesn't exist or is older than `file1.c`.

The format of Real-Time Workshop template makefiles follows the above example. Our template makefiles use additional features of `make` such as macros and file pattern matching expressions. In most versions of `make`, a macro is defined via

`MACRO_NAME =` *value*

References to macros are made via `$(MACRO_NAME)`. When `make` sees this form of expression, it substitutes `$(MACRO_NAME)` with *value*.

Pattern matching expressions are used to make the dependency rules more general. For example, using GNU Make we could have replaced the two "`file1.o: file1.c`" and "`file2.o: file2.c`" rules with the single rule:

```
%.o : %.c
        cc -c $<
```

The `$<` is a special macro that equates to the dependency file (i.e., `file1.c` or `file2.c`). Thus, using macros and the "%" pattern matching character, the above example can be reduced to:

```
SRCS = file1.c file2.c
OBJS = $(SRCS:.c=.o)

test: $(OBJS)
        cc -o $@ $(OBJS)

%.o : %.c
        cc -c $<
```

We generate the list of objects (`OBJS`) from the list of sources (`SRCS`) by using the string substitution feature for macro expansion. Here we replace the source file extension (.c) with the object file extension (.o). We also generalized the build rule for our program, `test`, to use the `$@` special macro that equates to the name of the current dependency target, in this case `test`.

## Generic Real-Time Templates

The Real-Time Workshop includes a set of built-in template makefiles that are set up for generic real-time code generation. These template makefiles allow you to simulate your fixed-step models on your workstation. This section discusses these template makefiles:

- `grt_unix.tmf` – targets UNIX platforms using any ANSI C compiler (`cc` is the default on all platforms except SunOS, where `gcc` is the default).
- `grt_vc.tmf` — creates a generic real-time executable for Windows 95 or Windows NT using Microsoft Visual C/C++.
- `grt_msvc.tmf` — creates a generic real-time project makefile for Windows 95 or Windows NT for use with Microsoft Visual C/C++.
- `grt_watc.tmf` — creates generic real-time executable for Windows 95 or Windows NT using the Watcom C/C++ compiler.

### grt_unix.tmf

The generic real-time template makefile for UNIX platforms is designed to be used with GNU Make. This makefile is set up to conform to the guidelines specified in the IEEE Std 1003.2-1992 (POSIX) standard.

You can supply the following options to `grt_unix.tmf` via arguments to the `make` command, `make_rtw`:

- `OPTS` — User specific options, such as `-DMULTITASKING` to enable multitasking mode, i.e.,:

  ```
  make_rtw OPTS="-DMULTITASKING"
  ```

- `OPT_OPTS` — Optimization options. Default is optimization option is `-O`. To turn off optimization and add debugging symbols, specify the `-g` compiler switch in the `make` command:

  ```
  make_rtw OPT_OPTS="-g"
  ```

- `USER_SRCS` — Additional user sources, such as files needed by S-functions. For example, suppose you have an S-function called `my_sfcn.c`, which is built with sfcn_lib1.c, and sfcn_lib2.c. (library routines for use with many S-functions). You can build my_sfcn.c using:

```
mex my_sfcn.c sfcn_lib1.c sfcn_lib2.c
```

In this case, the make command for the Real-Time Workshop should be specified as:

```
make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
```

- USER_INCLUDES — Additional include paths. For example, suppose you have two include paths (/appl/inc and /support/inc) that are required for the sfcn_lib1.c and sfcn_lib2.c files in the above example. The make command would be specified as:

```
make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
         USER_INCLUDES="-I/appl/inc -I/support/inc"
```

## grt_vc.tmf

The Real-Time Workshop provides a generic real-time template makefile (grt_vc.tmf) to create an executable for Windows 95 and Windows NT using Microsoft Visual C/C++. This template makefile is designed to be used with nmake, which is bundled with MicroSoft's Visual C/C++.

You can supply these options to grt_vc.tmf via arguments to the make command, make_rtw:

- OPTS — User-specific options, such as -DMULTITASKING to enable multitasking mode, i.e.:

```
make_rtw OPTS="-DMULTITASKING"
```

- OPT_OPTS — grt_vc.tmf optimization options. The default optimization option is -Oxat. To turn off optimization and add debugging symbols, specify the -Zd compiler switch in the make command:

```
make_rtw OPT_OPTS="-Zd"
```

- USER_SRCS — Additional user sources, such as files needed by S-functions. For example, suppose you have an S-function called my_sfcn.c, which is built with `sfcn_lib1.c`, and `sfcn_lib2.c`.(library routines for use with many S-functions). You can build `my_sfcn.c` using:

```
mex my_sfcn.c sfcn_lib1.c sfcn_lib2.c
```

In this case, the `make` command for the Real-Time Workshop should be specified as:

```
make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
```

- `USER_INCLUDES` — Additional include paths for example suppose you have two include paths (`c:\appl\inc` and `c:\support\inc`) that are required for the `sfcn_lib1.c` and `sfcn_lib2.c` files in the above example. The `make` command would be specified as:

```
make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
         USER_INCLUDES="-Ic:\appl\inc -Ic:\support\inc"
```

### grt_msvc.tmf

The Real-Time Workshop provides a generic real-time template makefile (`grt_msvc.tmf`) to create a Microsoft Visual C/C++ project makefile called *`model`*`.mak` for Windows 95 and Windows NT. `grt_msvc.tmf` is designed to be used with `nmake`, which is bundled with MicroSoft's Visual C/C++.

You can supply the following options to `grt_msvc.tmf` via arguments to the `make` command, `make_rtw`:

- `OPTS` — User-specific options, such as `/D MULTITASKING` to enable multitasking mode, i.e.:

```
make_rtw OPTS="/D MULTITASKING"
```

- `USER_SRCS` — Additional user sources, such as files needed by S-functions. For example, suppose you have an S-function called `my_sfcn.c`, which is built with `sfcn_lib1.c`, and `sfcn_lib2.c`. (library routines for use with many S-functions). You can build, `my_sfcn.c` using:

```
mex my_sfcn.c sfcn_lib1.c sfcn_lib2.c
```

In this case, the `make` command for the Real-Time Workshop should be specified as:

```
make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
```

- `USER_INCLUDES` — Additional include paths. For example suppose you have two include paths (`c:\appl\inc` and `c:\support\inc`) that are required for the sfcn_lib1.c and sfcn_lib2.c files in the above example. The make command would be specified as:

```
make_rtw USER_SRCS="sfcn_lib1.c sfcn_lib2.c"
         USER_INCLUDES="/I c:\appl\inc /I c:\support\inc"
```

### grt_watc.tmf

The Real-Time Workshop provides a generic real-time template makefile (grt_watc.tmf) to create an executable for Windows 95 and Windows NT using Watcom C/C++. This template makefile is designed to be used with nmake, which is bundled with Watcom C/C++.

The following options can be supplied to grt_watc.tmf via arguments to the make command, make_rtw:

- OPTS — User specific options, such as -DMULTITASKING to enable multitasking mode, i.e.:

  ```
  make_rtw OPTS="-DMULTITASKING"
  ```

- OPT_OPTS — grt_vc.tmf optimization options. This option is not used by grt_msvc.tmf. The default optimization option is -oxat. To turn off optimization and add debugging symbols, specify the -d2 compiler switch in the make command:

  ```
  make_rtw OPT_OPTS="-d2"
  ```

- USER_OBJS — Additional object (.obj) files, which are to be created from user sources, such as files needed by S-functions.For example, suppose you have an S-function called my_sfcn.c, which is built with sfcn_lib1.c, and sfcn_lib2.c. (library routines for use with many S-functions). You can build, my_sfcn.c using:

  ```
  mex my_sfcn.c sfcn_lib1.c sfcn_lib2.c
  ```

  In this case, the make command for the Real-Time Workshop should be specified as:

  ```
  make_rtw USER_OBJS="sfcn_lib1.obj sfcn_lib2.obj"
  ```

- USER_PATH — The directory path to the source (.c) files, which are used to create any .obj files specified in USER_OBJS. Multiple paths must be separated with a semicolon (e.g. USER_PATH="*path1*;*path2*"). For example, suppose `sfcn_lib1.c` exists in your local directory and `sfcn_lib2.c` exists in `c:\appl\src`, then the `make` command would be specified as:

```
make_rtw USER_OBJS="sfcn_lib1.obj sfcn_lib2.obj"
         USER_PATH="c:\appl\src"
```

- `USER_INCLUDES` — Additional include paths. For example, suppose you have two include paths (`c:\appl\inc` and `c:\support\inc`) that are required for the `sfcn_lib1.c` and `sfcn_lib2.c` files in the above example. The `make` command would be specified as:

```
make_rtw USER_OBJS="sfcn_lib1.obj sfcn_lib2.obj"
         USER_PATH="c:\appl\src"
         USER_INCLUDES="-Ic:\appl\inc -Ic:\support\inc"
```

# External Mode, Data Logging, and Signal Monitoring

## Introduction

When using the generated code in a rapid prototyping environment, you can modify the block parameters and log block outputs, system outputs, or inputs to specific blocks. If you are using an existing rapid prototyping environment, support has been added to allow you to tune parameters and log data.

To tune parameters, your rapid prototyping environment must support *External mode*, which refers to a simulation mode of Simulink. In External mode, Simulink waits for parameter changes. Once Simulink receives the parameter changes, it provides the target with the new parameters.

To observe and log how your code is executing, the Real-Time Workshop provides a C application program interface (API) that allows you to access block outputs. Note that this C API is not the MATLAB API.

This chapter discusses these topics:

- External mode
- How to use the Transmission Control Protocol (TCP) socket-based external mode implementation that is included in Simulink
- Implementing external mode interprocess communication using other protocols
- the API functions used for external mode implementation (reference pages are included)
- Parameter tuning by means of the `Parameters` data structure
- MAT-file data logging — you can log variables into a MAT-file for later analysis in MATLAB or for comparison with results from SIMULINK.
- Signal monitoring — you can access the outputs of individual blocks during program execution using the `BlockIOSignals` data structure

Before reading this chapter, you should have read Chapter 3, “Code Generation and the Build Process.” If you plan on implementing your own version of external mode for a custom or new real-time target, you need to know how MEX-files work (see the *MATLAB Application Program Interface Guide* for more information).

# External Mode

Simulink external mode is a mechanism that manages communication between Simulink and stand-alone programs built with the Real-Time Workshop. This mechanism allows you to use a Simulink block diagram as a graphical front end to the corresponding program (i.e., the program built from code generated for that block diagram).

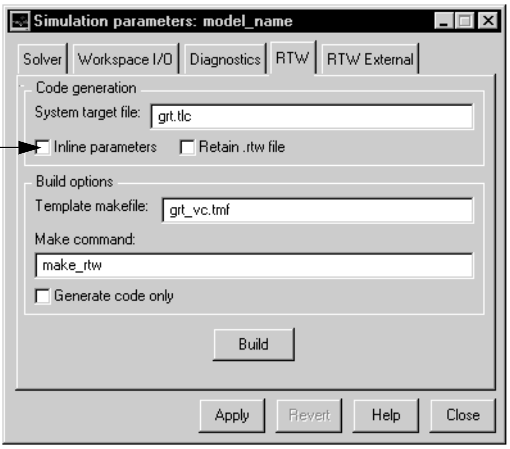
In external mode, whenever you change parameters in the block diagram, Simulink automatically downloads them to the executing program. This feature lets you perform parameter tuning in your program without recompiling.

External mode makes use of the client/server model of computing, where Simulink is a client that sends a request to the server (the external program) to install new parameter values. Structuring external mode in such a way makes it extensible to various protocols.

## Generating Code for Use with External Mode

Note that, when using external mode, you cannot generate code with program parameters embedded in the code (selected by using the **Inline parameters** button on the RTW page of the **Simulation parameters** dialog box). Ensure that this option is *not* selected when you generate code:

Do not select this option


Simulation parameters: model_name

Code generation
System target file: grt.tlc
Inline parameters
Retain .rtw file

Build options
Template makefile: grt_vc.tmf
Make command: make_rtw
Generate code only

## External Mode Operation

When external mode is enabled, Simulink does not simulate the system represented by the block diagram. Instead, it downloads initial parameters as soon as you start external mode. After the initial download, Simulink remains in a waiting mode until you change parameters in the block diagram. It then downloads the model parameters to the executing program.

## The Download Mechanism

When you change a parameter in the block diagram, Simulink calls the external link MEX-file, passing new parameter values (along with other information) as arguments. (MEX-files are subroutines that are dynamically linked to Simulink.)

The external link MEX-file contains code that implements one side of the interprocess communication (IPC) link. This link connects the Simulink process (where the MEX-file executes) to the process that is executing the external program.

The MEX-file transfers the new parameter values via this link to the external program. The other side of the communication link is implemented within the external program. This side writes the new parameter values into the program's `SimStruct` (the data structure containing all data relating to the model code). The section “Implementing the IPC Link” on page 4-15 provides more details on this process.

The Simulink side initiates the parameter download operation by calling a procedure on the external program side. In the general terminology of client/server computing, this means the Simulink side is the client and the external program is the server. The two processes can be remote, in which case a communication protocol is used to transfer data, or they can be local and employ shared memory to transfer data.

The following diagram illustrates this relationship:

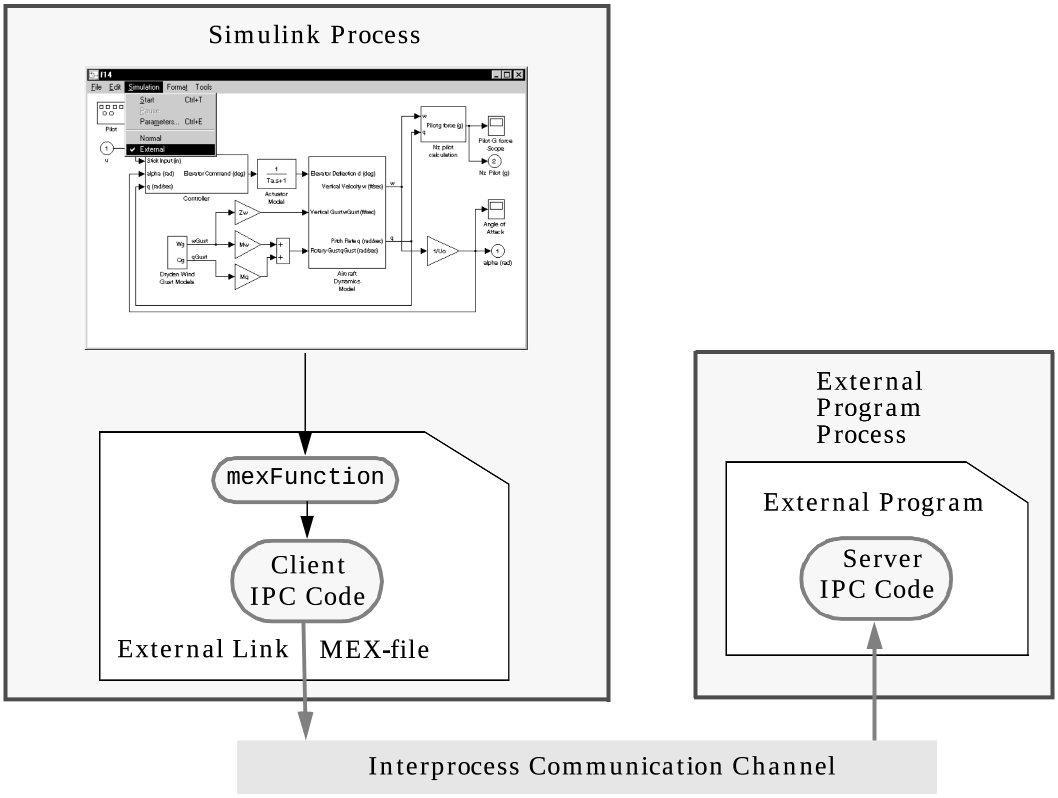


**Figure 4-1: External Mode Architecture**

Simulink calls the external link MEX-file whenever you change parameters in the block diagram. The MEX-file then downloads the parameters to the external program via the communication channel.

## Limitations

In general, you cannot change a parameter if doing so results in a change in the structure of the model. For example, you cannot change:

- The number of states, inputs, or outputs of any block
- The sample time or the number of sample times
- The integration algorithm for continuous systems
- The name of the model or of any block
- The parameters to the Fcn block

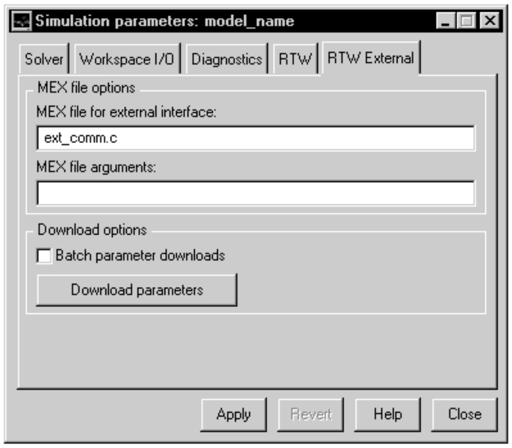
If you cause any of these changes to the block diagram, then you must rebuild the program with newly generated code.

However, parameters in transfer function and state space representation blocks *can* be changed in specific ways:

- The parameters (numerator and denominator polynomials) for the Transfer Fcn (continuous and discrete) and Discrete Filter blocks can be changed (as long as the number of states does not change).
- Zero entries in the State Space and Zero Pole (both continuous and discrete) blocks in the user-specified or computed parameters (i.e., the A, B, C, and D matrices obtained by a zero-pole to state-space transformation) cannot be changed once external simulation is started.
- In the State Space blocks, if the user specifies the matrices in the "controllable canonical realization," then all changes to the A, B, C, D matrices that preserve this realization and the dimensions of the matrices are allowed. For a discussion of controllable canonical realizations, see *Linear Systems Theory* by C.T. Chen.

## External Mode Configuration

You can use external mode with your target, providing an external link MEX-file exists that communicates with your target system. To configure external mode, open the RTW External page of the **Simulation parameters** dialog box:



MEX file options
MEX file for external interface: ext_comm.c
MEX file arguments:
Download options
☐ Batch parameter downloads

You must set the MEX-file options:

- MEX-file for external interface - name of the external link MEX-file.
- MEX-file arguments - any arguments for the external link MEX-file. For example, the TCP-based `ext_comm` external link MEX-file described in the following sections of this chapter contains three optional arguments: the network name of your target, the verbosity level, and a TCP server port number.

The **Batch parameter downloads** check box enables/disables batch parameters changes. When batch mode is enabled, changes made to block parameters are sent only after the **Download parameters** button is clicked. If batch mode is not enabled, changes made to block parameters are sent immediately. You can change any block parameters that are MATLAB workspace variables by modifying the variable in the MATLAB workspace and clicking the **Download parameters** button.

## TCP Implementation

The Real-Time Workshop provides code to implement both the client and server side based on Transmission Control Protocol (TCP). You can use socket-based external mode implementation provided by the Real-Time

Workshop with the generated code, provided that your target system has an external mode server. For example, the Tornado environment and generic real-time targets contain an external server.

The following section discusses how to use external mode with real-time programs on a UNIX system. Chapter 8, “Targeting Tornado for Real-Time Applications.” illustrates the use of external mode in the Tornado environment.

## Using the TCP Implementation

This section describes how to use the TCP-based client/server implementation provided with the Real-Time Workshop. If you want to write your own interprocess communication code based on other protocols, see “Implementing the IPC Link” on page 4-15.

In order to use Simulink external mode, you must:

- Specify the name of the external link MEX-file in the RTW external page of the **Simulation parameters** dialog box.
- Compile the external link MEX-file using the `mex` command and copy the executable to the current directory or a directory that is on your MATLAB path.
- Configure the template makefile so that it links the proper source files for the TCP server code and defines the necessary compiler flags when building the generated code.
- Build the external program.
- Run the external program.
- Set Simulink to external mode and start the simulation.

This figure shows the structure of the TCP-based implementation:

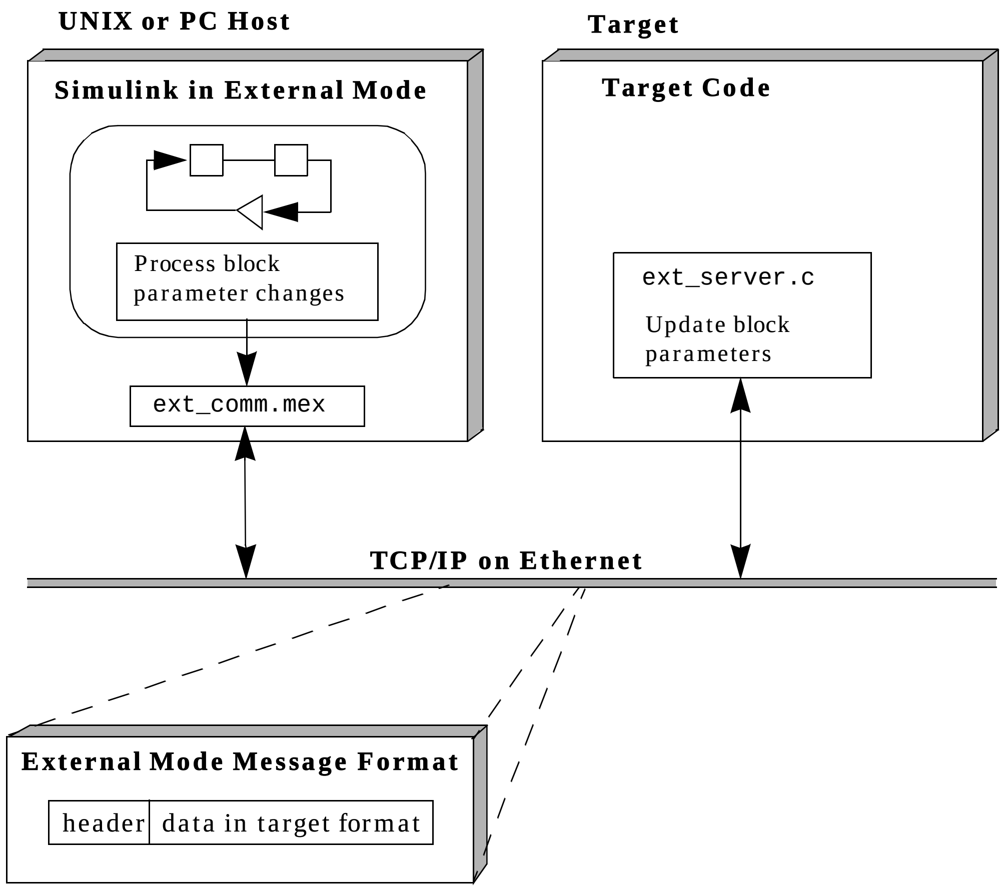
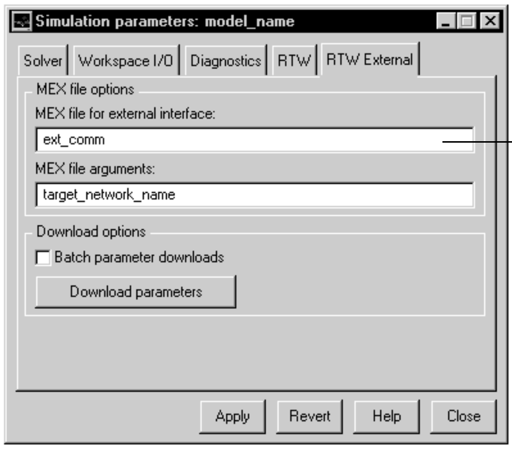


**Figure 4-2: TCP-based Client/Server Implementation for External Mode**

The following sections discuss the details of how to use the external mode of Simulink.

## The External Link MEX-File

You must specify the name of the external link MEX-file in the RTW External page of the **Simulation parameters** dialog box:



MEX file for external interface: ext_comm

MEX file arguments: target_network_name

Enter the name of the external link interface MEX-file in the box (you do not need to enter the `.mex` extension). This file must be in the current directory or in a directory that is on your MATLAB path.

You may be required to provide arguments to the MEX-file. In particular, `ext_comm` has three optional arguments: the network name of your target, the verbosity level, and the TCP port number. The implementation of the TCP-based MEX-file is called `ext_comm`, which is the default for this dialog box.

### MEX-File Optional Arguments

You can specify optional arguments in the RTW External page of the simulation parameters dialog box that are passed to the MEX-file. These include the name of the target host, the verbosity level, and optionally the TCP server port number:

1. Target network name — The network name of the computer running the external program. By default, this is the computer on which Simulink is running.
2. Verbosity level — Controls the level of detail of the information printed out during the actual data transfer. The value is either 0 or 1 and has the following meaning:

   0 — no information
   1 — detailed information
3. TCP server port number — The default value is 17725. You can change the port number to a value between 245 and 65535 to avoid a port conflict if necessary.

You must specify these options in order. For example, if you want to specify the verbosity level (the second argument), then you must also specify the target host name (the first argument).

Note that you can also specify verbosity level and port number as options in the external program. See "Running the External Program" on page 4-13 for more information.

## External Mode Support

The generic real-time and Tornado targets have support for external mode. To enable external mode, specify

```
make_rtw EXT_MODE=1
```

in the make command file of the RTW page of the **Simulation parameters** dialog box.

## Building the External Program

Once you have modified the make command, click the **Build** button to build the program. You can use the MAT-file data logging options to observe the effects of changing parameters in a real-time program. Tornado applications can also use StethoScope with signal monitoring to observe the effect of changing parameters.

See Chapter 2, “Getting Started with the Real-Time Workshop,” for an example of how to build the program and save data that you can display with MATLAB. See Chapter 8, “Targeting Tornado for Real-Time Applications,” for information on building programs for Tornado and using StethoScope.

## Running the External Program

The external program must be running before you can use Simulink in external mode.

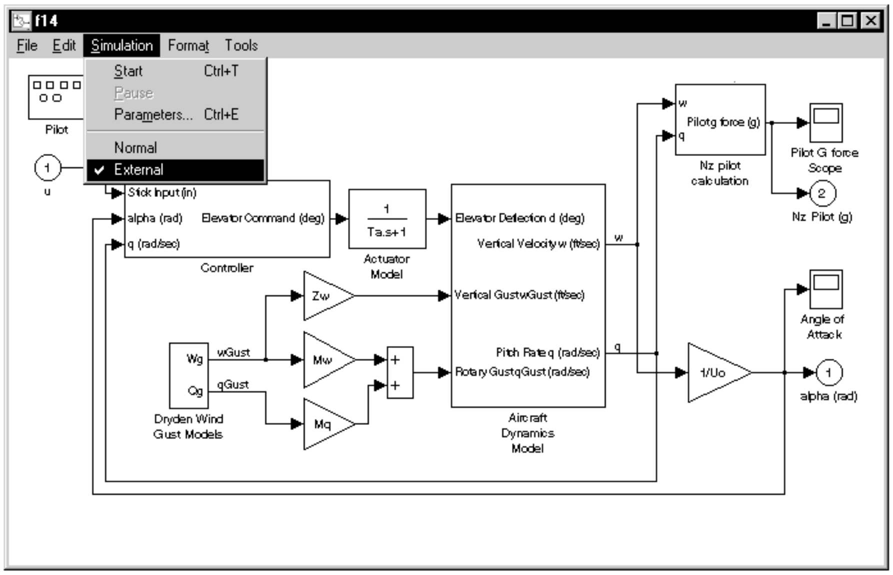
In the UNIX environment, if you start the external program from MATLAB, you must run it in the background so that you can still access Simulink. For example, the command:

```
!model &
```

runs the executable file `model` from MATLAB by spawning another process to run it.

## Enabling External Mode

To enable external mode, display the Simulink block diagram and select **External** from the **Simulation** menu:



Next, select **Start** from the **Simulation** menu, as you would to begin a simulation and change parameters as desired.

### Possible Error Conditions

If the Simulink block diagram does not match the external program, Simulink displays an error box informing you that the checksums do not match (i.e., the model has changed since you generated code). This means you must rebuild the program from the new block diagram (or reload the correct one) in order to use external mode.

If the external program is not running, Simulink displays an error informing you that it cannot connect to the external program.

## Implementing the IPC Link

This section provides information about how to implement interprocess communication (IPC) using a protocol other than TCP.

### System Components

Simulink external mode requires the following components:

- Simulink running in external mode, displaying the block diagram from which you generated the external program.
- A MEX-file that implements the client side of the IPC link.
- An external (i.e., a stand-alone executable) program built from the code generated for the Simulink model. This program must contain code that implements the server side of the IPC link.

#### The Client-Side Implementation

The Real-Time Workshop defines an API for implementing the external link MEX-file (i.e., the client side of the IPC link). This API defines function prototypes and a data structure that is used to transfer parameter values (and other required information) from Simulink to the MEX-file.

The external link MEX-file must define the following three functions:

**Table 4-1: API Routines Defined by the Real-Time Workshop**

| Function | Description |
|---|---|
| `mdlCommInitiate` | Open a communication channel to the external program. |
| `mdlSetParameters` | Download parameters to the external program. |
| `mdlCommTerminate` | Close the communication channel opened with the `mdlCommInitiate` function. |

See pages 4-17 through 4-19 for more information on these functions.

**Left-Hand Side Arguments.** When Simulink calls the external link MEX-file, it passes a pointer to a predefined data structure as the left-hand side argument.

This data structure contains the new parameter values as well as other data required by the three functions listed in the table above.

The MEX-file entry point function, `mexFunction`, calls each of the required functions, which in turn pass the data to the external program. `mexFunction` and the data structure are defined in the file `ext_main.c` and `extsim.h` respectively. This file resides in the *matlabroot*`/ext_mode` directory and must be included at the end of your external mode MEX-file.

You must write the code that implements the required functions listed in the table above. The sample TCP-based implementation defines these functions in the file, `ext_comm.c`, which is in the *matlabroot*`/ext_mode` directory.

To simplify your implementation, the Real-Time Workshop includes a template version of `ext_comm.c` (called `ext_tmpl.c`) that contains prototypes for each function. It is in the *matlabroot*`/ext_mode` directory.

**Right-Hand Side Arguments.** The MEX-file right-hand side arguments are used to pass optional arguments to the MEX-file. In the TCP implementation, there are three optional arguments — target network name, verbosity level, and TCP server port number. However, your implementation can use whatever set of parameters are appropriate for it.

### Server Side Implementation

The way in which you implement the server code (i.e., the code linked to the external program) depends on the IPC communication mechanism. However, regardless of the technique used to transfer data between the two processes, the ultimate result must be to change block parameter values in the external program's `SimStruct`. You can do this by obtaining a pointer to the parameter vector in the real-time code, using the `SimStruct` access macro, `ssGetDefaultParam`.

The *matlabroot*`/c/src` directory contains the source files for the sample server-side TCP implementation. These files are called `ext_svr.c`, `ext_svr.h`, and `ext_msg.h`.

**Purpose**

Establishes a connection between Simulink and an external program.

**Synopsis**

```
int mdlCommInitiate(int block_param_count, double *block_params,
                    char *model_name, int model_checksum, int nrhs,
                    mxArray *prhs[], char *error_message)
```

**Arguments**

| | |
|---|---|
| `block_param_count` | Total number of block parameters |
| `block_params` | Vector of new parameter values |
| `model_name` | Name of the Simulink model |
| `model_checksum` | Simulink model checksum |
| `nrhs` | Number of additional MEX-file arguments |
| `prhs` | Additional MEX-file arguments |
| `error_message` | Error message returned to Simulink |

**Description**

This function is called by the external link MEX-file to establish a connection between the Simulink process and the external program process. It generally performs the following operations:

- Initilizes the communications channel
- Compares the Simulink model name and model checksum with that of the external program to ensure that the Simulink block diagram matches the code in the external program
- Parses any additional arguments passed to the MEX-file
- Generates an error message, if an error occurs

This function is called only once. Use `mdlSetParameters` to download new values.

**Returns**

0 if successful or –1 if an error occurs

## mdlSetParameters

**Purpose** Downloads parameters from Simulink to the external program.

**Synopsis**

```
int mdlSetParameters(int block_param_count, double *block_params,
                     int num_changed, int *elems, char *model_name,
                     int model_checksum, int nrhs, mxArray *prhs[],
                     char *error_message)
```

**Arguments**

| | |
|---|---|
| `block_param_count` | Total number of block parameters |
| `block_params` | Vector of new parameter values |
| `num_changed` | Number of parameters that changed |
| `elems` | Indices of elements in the block_params vector that have changed |
| `model_name` | Name of the Simulink model |
| `model_checksum` | Simulink model checksum |
| `nrhs` | Number of additional MEX-file arguments |
| `prhs` | Additional MEX-file arguments |
| `error_message` | Error message returned to Simulink |

**Description** This function is called when parameters are changed in the Simulink block diagram. Simulink passes it a vector of block parameters along with a count of the number that have changed and vector of the indices of the elements in the parameter vector that correspond to the changed parameters.

`mdlSetParameters` downloads to the external program only the parameters that have changed.

**Returns** 0 if successful or –1 if an error occurs

**Purpose**

Closes the communication link between Simulink and the external program.

**Synopsis**

```
int mdlCommTerminate(char *model_name, int model_checksum, int nrhs,
                     mxArray *prhs[], char *error_message)
```

**Arguments**

| | |
|---|---|
| `model_name` | Name of the Simulink model |
| `model_checksum` | Simulink model checksum |
| `nrhs` | Number of additional MEX-file arguments |
| `prhs` | Additional MEX-file arguments |
| `error_message` | Error message returned to Simulink |

**Description**

This function closes the communication link established by the `mdlCommInitiate` function.

**Returns**

0 if successful or –1 if an error occurs

## Data Logging and Signal Monitoring

The Real-Time Workshop provides two mechanisms for feedback from your external simulation, MAT-file data logging and signal monitoring.

### MAT-File Data Logging

The external program can create a MAT-file that logs system states and outputs at each simulation time step. By using the Workspace I/O page of the **Simulation parameters** dialog box, you can select time, state and outputs to log as well as the decimation desired. The variable names used in the MAT-file are `rt_tout`, `rt_xout` and `rt_yout` respectively. The Real-Time Workshop logs outputs for

- All root Outport blocks (rt_yout)
- All Scopes that have ‘save data to workspace’ selected
- All To Workspace blocks in the model

For Scope and To Workspace blocks, you must specify variable names in each block’s dialog box. You can log states for all continuous and discrete states in the model. The sort order of the `rt_yout` array is based on the port number of the Outport block, starting with 1. You can determine the sort order of the rt_xout array by using this MATLAB command:

```
[a,b,c,d] = model([],[],[],0)
```

and inspecting the variable *c*. The filename defaults to *model*`.mat` but can be changed by specifying `OPTS="-DSAVEFILE=filename"` in the Make command field on the RTW page of the **Simulation parameters** dialog box.

The system target file (for example, `grt.tlc`) includes the statement:

```
%assign MatFileLogging = 1
```

This enables Scope and To Workspace blocks to log data. Outport and To File blocks are not affected by the flag.

By default, MAT-file data logging is performed in generic real-time and DOS targets by calling the logging function `rt_UpdateTXYLogVars` at every simulation step in `grt_main.c` or `drt_main.c` respectively. You can optionally include it in the `rt_main.c` file of Tornado targets by specifying `MAT_FILE=1` in the Make command field on the RTW page.

## To File Block MAT-Files

Aside from the Workspace I/O MAT-file logging described above, MAT-file logging can be done using the To File block. A separate MAT-file containing time and input variable(s) is created for every To File block in the model. You must specify the filename, variable name, decimation and sample time in the To File block's dialog box. The To File block cannot be used in DOS real-time targets because of limitations of DOS.

## Signal Monitoring

Signal Monitoring provides a second method for accessing block outputs in an externally running program. All block output data is written to the SimStruct with each time step in the model code. However, to access the output of any given block in the `SimStruct`, you must know the index into the `BlockIO` vector where the data is stored, how many output ports the block has, and the width of each output. All of this information is contained in the `BlockIOSignals` data structure. This is the mechanism that the StethoScope Graphical Monitoring / Data Analysis Tool uses to collect signal information in Tornado targets. See Chapter 8, " Targeting Tornado for Real-Time Applications," for more information on using StethoScope.

The `BlockIOSignals` data structure is created during code generation only if you include

```
    %assign BlockIOSignals = 1
```

in the system target file (for example, `tornado.tlc`). Otherwise, `BlockIOSignals` is not created. If included, the TLC file *matlabroot*`/rtw/c/tlc/biosig.tlc` creates the file *model*`.bio` that contains the information on all block outputs in the model in an array of structures. This array can then be used to monitor the value of any output while the program is running. The structure definition for the elements of the array is in *matlabroot*`/rtw/c/src/bio_sig.h`. This file is included by *model*`.bio` and should also be included by any source file that uses the array (for example, `rt_main.c`). Note that, depending on the size of your model, the `BlockIOSignals` array can consume a considerable amount of memory.

## Using BlockIOSignals to Obtain Block Outputs

The `BlockIOSignals` data structure is declared as follows:

```
typedef struct BlockIOSignals_tag {
  char_T *blockName;    /* Block's full path name */
  char_T *signalName;   /* Signal label (NULL if not present) */
  uint_T portNumber;    /* Block output port # (starting at 0) */
  uint_T signalWidth;   /* Signal's width */
  uint_T signalOffset;  /* Signal's offset in block I/O vector */
} BlockIOSignals;
```

You can obtain the base address to which the `SignalOffset` should be added by using the `Simstruct` access macro `ssGetBlockIO`.

The model code file *model*`.bio` defines an array of `BlockIOSignals` structures, for example:

```
#include "bio_sig.h"

/* Block output signal information */
const BlockIOSignals rtBlockIOSignals[] =
{
  /* blockName signalName portNumber signalWidth signalOffset */
  {
    "simple/Constant1",
    NULL, 0, 1, 0
  },
  {
    "simple/Constant",
    NULL, 0, 1, 1
  },
  {
    "simple/Gain",
    "accel", 0, 2, 2
  },
  {
    NULL, NULL, 0, 0, 0
  }
};
```

Each structure element describes one output port for a block. Thus, a given block will have as many entries as it has output ports. In the above example, the block simple/Gain has a signal named `accel` on block output port 0. The width of the signal is 2 and it starts at offset 2 in the `BlockIO` vector.

The array is accessed via the name `rtBlockIOSignals` by whatever code would like to use it. The `SimStruct` access macro `ssGetNumBlockIO` can be used to determine the number of elements in the array or the fact that last element has a `blockName` of `NULL` could be used. You should then write code that walks through the `rtBlockIOSignals` array and chooses the signals to be monitored based on the `blockName` and `signalName` or `portNumber`. How the signals is monitored is up to you. For example, the signals could be collected every time step or just sampled asynchronously by a separate, lower priority task.

For example, the Tornado source file, `rt_main.c`, defines the following function that selectively installs signals from the `BlockIOSignals` array into the Stethoscope Data Analysis tool by calling `ScopeInstallSignal`. The signals are then collect in the main simulation task by calling `ScopeCollectSignals`.

The code below is an example routine that installs signals from the BlockIOSignals array:

```
static void rtInstallRemoveSignals(SimStruct *S,
                 char_T *installStr, int_T fullNames, int_T flag)
{
  register int_T         i, w;
  real_T                 *blockIOVector = ssGetBlockIO(S);
  char_T                 *blockName;
  char_T                 name[1024];
  extern BlockIOSignals  rtBlockIOSignals[];

/* Make sure that Stethoscope has been properly initialized. */
  ScopeInitServer(4*32*1024, 0, 0);

  if (installStr == NULL) {
    return;
  }

  for(i = 0; i < ssGetNumBlockIO(S); i++) {
    BlockIOSignals *blockInfo = &rtBlockIOSignals[i];
    if (fullNames) {
      blockName = blockInfo->blockName;
    } else {
      blockName = strrchr(blockInfo->blockName, '/');
      if (blockName == NULL) {
        blockName = blockInfo->blockName;
      } else {
        blockName++;
      }
    }
    if ((*installStr) == '*') {
    } else if (strcmp("[A-Z]*", installStr) == 0) {
      if (!isupper(*blockName)) {
        continue;
      }
    } else {
      if(strncmp(blockName,installStr,strlen(installStr))!=0) {
        continue;
      }
    }
    /*install/remove the signals*/
    for (w = 0; w < blockInfo->signalWidth; w++) {
      sprintf(name,"%s_%d_%s_%d",blockName,blockInfo->portNumber,
       (blockInfo->signalName==NULL)?"":blockInfo->signalName, w);
      if (flag) { /*install*/
        if( ScopeInstallSignal(name, "units",
            &blockIOVector[blockInfo->signalOffset+w],"double",0)){
          fprintf(stderr,"rtInstallRemoveSignals:ScopeInstallSignal"
                  "possible error: over 256 signals.\n");
          return;
        } else { /*remove*/
          if (!ScopeRemoveSignal(name, 0)) {
            fprintf(stderr,"rtInstallRemoveSignals:
                    ScopeRemoveSignal\n""%s not found.\n",name);
          }
        }
      }
    }
  }
```

Below is an excerpt from an example routine that collects signals taken from the main simulation loop:

```
  /*******************************************
   * Step the model for the base sample time *
   *******************************************/
  MdlOutputs(FIRST_TID);

  #ifdef MAT_FILE
  if (rt_UpdateTXYLogVars(S) != NULL) {
          fprintf(stderr,"rt_UpdateTXYLogVars() failed\n");
          return(1);
      }
  #endif

  #ifdef STETHOSCOPE
      ScopeCollectSignals(0);
  #endif

  MdlUpdate(FIRST_TID);
  <code continues ...>
```

The Real-Time Workshop provides a mechanism that allows your application program to monitor block outputs during program execution.

All block output data is written to the `SimStruct` at each step through the model code. However, to access the output of any given block in the `SimStruct`, you must know the index into the `BlockIO` vector where the data is stored, as well as how many output ports the block has and the width of each output. All of this information is contained in the `ModelBlockInfo` data structure.

The `ModelBlockInfo` data structure is created by the generated code only if you define the flag `USE_MDLBLOCKINFO` when you build your program. Otherwise, `ModelBlockInfo` is not created.

Note that, depending on the size of your model, `ModelBlockInfo` can consume a considerable amount of memory.

# Model Code

## Introduction

This chapter describes architecture of the generated ANSI C code produced by the Real-Time Workshop from Simulink block diagrams. The generated code is essentially a set of procedures that are executed by a target-specific run-time interface. The generated code is highly optimized and produces the same numerical results as your Simulink block diagram.

This chapter describes the file contents of

- *model*`.c` — the model C source code
- *model*`.h` — the model header include file
- *model*`.prm` — the parameter definition include file
- *model*`.reg` — the model registration include file

Additionally, if you are using Stateflow charts in your Simulink model, the Real-Time Workshop generates *model_rtw*`.c` — the Stateflow C souce code. Refer to the Stateflow documentation for more information about this file.

In addition, this chapter discusses file splitting, an automatic procedure supported by the Real-Time Workshop where large models generate multiple files containing the model code.

All working data is stored in a Simulink structure named `SimStruct`. All data is typed according to `typedef`'s defined in `tmwtypes.h`, which is located in *matlabroot*`/extern/include`.

All global data symbols in the generated code start with `rt`. All global functions, other than the model functions such as `MdlOutputs`, start with `rt_`.

## model.c

The *model*.c file contains the procedures that implement the algorithm defined by your Simulink block diagram. The general format of this file is shown in Figure 5-1 on page 5-4. This code is designed for one instance of the model within a given program. It makes effective use of pre-allocated variables defined in *model*.prm. A Simulink block can be generalized to the following set of equations:

$$y = f_0(t, x_c, x_d, u)$$

Output, $y$, is a function of continuous state, $x_c$, discrete state, $x_d$, and input, $u$. Each block writes its specific equation in the appropriate section of `MdlOutput`.

$$x_{d+1} = f_u(t, x_d, u)$$

The discrete states, $x_d$, are a function of the current state and input. Each block that has a discrete state updates its state in `MdlUpdate`.

$$\dot{x} = f_d(t, x_c, u)$$

The derivatives, $\dot{x}$, are a function of the current input. Each block that has continuous states provides its derivatives to the solver (e.g., `ode5`) in `MdlDerivatives`. The derivatives are used by the solver to integrate the continuous state to produce the next value.

The output, y, is generally written to the block I/O structure. Root-level Outport blocks write to the external outputs structure. The continuous and discrete states are stored in the states structure. The input, $u$, can originate from another block's output, which is located in the block I/O structure, an external input (located in the external inputs structure), or a state. All structures are defined in *model*.h.

```
/*
 * Version, Model options, TLC options,
   and code generation information.
 */

/* Place any system includes here. */
#include "model.h"
#include "model.prm"

void MdlStart(void)
{
/*
  State initialization code.
  Model start-up code - one time initialization code.
  Execute any block enable methods.
  Initialize output of any blocks with constant sample times.
*/
}

void MdlOutputs(int_T tid)
{
/* Compute: y = f0(t,xc,xd,u) for each block as needed. */
}

void MdlUpdate(int_T tid)
{
/* Compute: xd+1 = fu(t,xd,u) for each block as needed. */
}

void MdlDerivatives(void)
{
/* Compute: dxc = fd(t,xc,u) for each block as needed. */
}

void MdlTerminate(void)
{
/* Perform shutdown code for any blocks that
   have a termination action */
}
#include "model.reg"
```

**Figure 5-1: Contents of model.c**

## void MdlStart(void)

After the model registration functions, `MdlInitializeSizes` and `MdlInitializeSampleTimes` (located in *model*`.reg`), have been executed, the run-time interface starts execution by calling `MdlStart`. This routine is called once at start-up.

The function `MdlStart` has four basic sections:

- Code to initialize the states for each block in the root model that has states. A subroutine call is made to the "initialize states" routine of conditionally executed subsystems.
- Code generated by the one-time initialization (start) function for each block in the model.

  Code to enable: 1) the blocks in the root model that have enable methods, 2) the blocks inside triggered or function-call subsystems residing in the root model. Simulink blocks can have enable and disable *methods*. An enable method is called just before a block starts executing, and the disable method is called just after the block stops executing.
- Code for each block in the model that has a constant sample time.

## void MdlOutput(int_T tid)

`MdlOutput` is responsible for updating the output of blocks at appropriate times. The `tid` (task id) parameter identifies the task that in turn maps when to execute blocks based upon their sample time. This routine is invoked by the run-time interface during major and minor time steps. The major time steps are when run-time interface is taking an actual time step (i.e., it is time to execute a specific task). If your model contains continuous states, the minor time steps will be taken. The minor time steps are when the solver is generating integration stages, which are points between major outputs. These integration stages are used to compute the derivatives used in advancing the continuous state.

## void MdlUpdate(int_T tid)

`MdlUpdate` is responsible for updating the discrete states and work vector state information (that is, states that are neither continuous nor discrete) saved in work vectors. The `tid` (task id) parameter identifies the task that in turn indicates which sample times are active allowing you to conditionally update states of only active blocks. This routine is invoked by the run-time interface after the major `MdlOutput` has been executed.

## void MdlDerivatives(void)

`MdlDerivatives` is responsible for returning the block derivatives. This routine is called in minor steps by the solver during its integration stages. All blocks that have continuous states have an identical number of derivatives. These blocks are required to compute the derivatives so that the solvers can integrate the states.

## void MdlTerminate(void)

`MdlTerminate` contains any block shutdown code. `MdlTerminate` is called by the run-time interface, as part of the termination of the real-time program.

## Conditionally Executed Subsystems

The block code for conditionally executed subsystems, (i.e., enabled, triggered, triggered and enabled, or function-call subsystems) is placed in separate procedures. These procedures are placed in the *model*`.c` file above `MdlOutputs`.

### Enabled Subsystems

The following routines are generated for enabled subsystems:

- `void Sys_`*subname*`_Initialize(void)` — This routine is created if the blocks within the subsystem have to initialize states.
- `void Sys_`*subname*`_Enable(void)` — This routine is created if any blocks within the subsystem need to know when the subsystem starts executing.
- `void Sys_`*subname*`_Disable(void)` — This routine is created if any blocks within the subsystem need to know when the subsystem must stop executing.
- `void Sys_`*subname*`_Output(int_T tid)` — This routine computes the output for blocks within the subsystem.
- `void Sys_`*subname*`_Update(int_T tid)` — This routine is created if any blocks within the subsystem need to update their discrete states.
- `void Sys_`*subname*`_Derivative(void)` — This routine is created if blocks within the subsystem have continuous states.

### Triggered Subsystems

The following routine is generated for triggered subsystems:

- `void Sys_`*subname*`_OutputUpdates(void)` — This routine does both block output and discrete state update in one call.

### Triggered and Enabled Subsystems

The following routines are generated for triggered and enabled subsystems:

- `void Sys_`*subname*`_Enable(void)` — This routine is created if any blocks with in the subsystem need to know when the subsystem starts executing.
- `void Sys_`*subname*`_Disable(void)` — This routine is created if any blocks within the subsystem need to know when the subsystem stops executing.
- `void Sys_`*subname*`_OutputUpdates(void)` — This routine does both block output and discrete state update in one call.

### Function-Call Subsystems

The following routine is generated for function-call subsystems:

- `void Sys_`*subname*`_OutputUpdates(void)` — This routine does both block output and discrete state update in one call.

Note that the function-call subsystem routine is never executed directly by any of the procedures within the generated code. The Real-Time Workshop packages the subsystem into a single unit so that Stateflow or any S-function block can execute the function-call subsystem directly.

## model.h

The model header file, *model*.h, contains structure definitions for:

- Block parameters
- Block I/O
- States
- State derivatives
- Block work areas (real, integer, and pointer)
- Block modes
- External inputs
- External outputs
- Previous zero crossings
- Data store memory

The data declarations for these structures are declared in *model*.prm. There is only one instance of these structures per model. Pointers to these structures are cached away in the SimStruct.

## Block Parameters Structure

The block parameters, for example, the gain value of a gain block, are placed in a structure resembling this code:

```
typedef struct Parameters_tag {
 struct {
    real_T parameter_1;
    real_T parameter_i;
    real_T parameter_n;
  } blockName_1;
   :
  struct {
    real_T scalarParam;
    real_T vectorParam[n];
    real_T matrixParam[nRows][nCols];  Non S-function
                   matrix is defined
                                       as [nRows][nCols]
 } nonSfunctionBlock;
   :
  struct {
    real_T scalarParam;
    real_T vectorParam[n];
    real_T matrixParam[nCols][nRows];  S-function
                   matrix is defined
                                       as [nCols][nRows]
  } sfunctionBlock;
   :
} Parameters;
```

Parameter values will have an optional `[vector_length]` or `[nRows][nCols]` for vectors and matrix parameters, respectively. A subtle note is that an S-function matrix will be declared by `[nCols][nRows]`. This is done in order to maintain consistency with MATLAB.

The block parameters structure is eliminated when the `.rtw` file identifier, `NumBlockParams`, is 0 (i.e., your model doesn't contain blocks with parameters, or when inlining parameters is selected and none of the parameter values requires persistence (i.e., when parameters are accessed in a manner that requires them to reside in memory).

When inlining parameters the goal is to eliminate the parameters structure entirely and inline values directly into the source code. However, this is not possible for:

- Noninlined S-function parameters — the S-function API requires access to the parameters, through routines such as mxGetPr. By inlining your S-function, however, you can eliminate the parameters if they are not used duing a vectored roll operation.
- A block vector parameter, and the block will roll (i.e., the generated code is placed in a "for" loop.

If the model contains any of these parameters, the parameters structure persists, but it contains only these parameters. All other parameters are eliminated from the parameters structure.  That is, scalars and vector parameters in a block that does not roll are eliminated from the parameters structure, and their values are inserted directly in the generated code.

## Block I/O Structure

The block I/O structure consists of all block output signals. The number of block output signals is the sum of the widths of the data output ports of all nonvirtual blocks in your model. Structure field names are determined by either the block's output signal name, when present, or by the block name and port number when the output signal is left unlabeled. The general format of the block I/O structure is:

```
typedef struct BlockIO_tag {
  real_T BlockOutputSignal_1;
  real_T BlockOutputSignal_j;
  real_T BlockOutputSignal_n;
} BlockIO;
```

Block output signals have an optional [vector_length] for wide signals.

## States Structure

The states structure contains the continuous and discrete state information for any blocks in your model that have states. The states structure has two sections: the first is for the continuous states, and the second is for the discrete states. This is the code for the states structure:

```
typedef struct States_tag {
  struct {
    real_T block_1;
    real_T block_i;
    real_T block_n;
  } c;             Continuous states, only present if NumContStates >
0
struct {
    real_T block_1;
    real_T block_i;
    real_T block_n;
  } d;             Discrete states, only present if NumDiscStates > 0
} States;
```

Note that the derivatives of the continuous states are managed by the solver. Thus, to access the derivatives vector in an S-function you need to use the `SimStruct` `ssGetdX` access methods.

## State Derivatives Structure

The state derivatives structure contains the continuous state derivative information for any blocks in your model that have continuous states. This is the general format for the states derivative structure:

```
typedef struct StatesDerivatives_tag {
  real_T block_1;
  real_T block_i;
  real_T block_n;
} StatesDerivatives;
```

Block with multiple continuous states will have `[vector_length]` appended.

Note that the derivatives of the continuous states are managed by the solver. Thus, to access the derivatives vector in an S-function you need to use the `SimStruct` `ssGetdX` access methods.

## Real Work Vector Structure

Blocks may have a need for real work areas. For example, the memory block uses a real work element for each signal. Real work areas are used to save real state information when the type of state doesn't correspond to continuous or discrete states. The general format of the real work structure is:

```
typedef struct R_Work_tag {
  struct {
    real_T rwork_1;
    real_T rwork_i;
    real_T rwork_n;
  } blockName_1;
  :
  struct {
    real_T rwork_1;
    real_T rwork_i;
    real_T rwork_n;
  } blockName_n;
  real_T sfunction1;
  real_T sfunction2;
} R_Work;
```

Blocks have an optional `[vector_length]` when more than one element is required for the block. This structure is created only when the number of real work elements for all blocks in the model is greater than zero.

By default, an S-function real work area is defined as a single contiguous vector. However, you can change the S-function definition using the TLC function `LibDefineBlockRWork` inside the S-function's TLC file (actually, you can redefine any block's real work elements using this function). See the *Target Language Compiler Reference Guide* for more information about naming real work elements for an S-function.

## Integer Work Vector Structure

Blocks may have need for integer work areas. Integer work areas are used to save state information when the type of state doesn't correspond to continuous or discrete states. The general format of the integer work vector structure is:

```
typedef struct I_Work_tag {
  struct {
    int_T iwork_1;
    int_T iwork_i;
    int_T iwork_n;
  } blockName_1;
  :
  struct {
    int_T iwork_1;
    int_T iwork_i;
    int_T iwork_n;
  } blockName_n;
  int_T sfunction1;
  int_T sfunction2;
} I_Work;
```

Blocks have an optional `[vector_length]` when more than one element is required for the block. This structure is created only when the number of integer work elements for all blocks in the model is greater than zero.

By default, an S-function integer work area is defined as a single contiguous vector. However, you can change the S-function definition using the TLC function `LibDefineBlockIWork` inside the S-function's TLC file (actually, you can redefine any block's integer work elements using this function). See the *Target Language Compiler Reference Guide* for more information about naming integer work elements for an S-function.

## Pointer Work Vector Structure

Blocks may have need for pointer work areas. Pointer work areas are used to save state information when the type of state doesn't correspond to continuous or discrete states. The general format of the pointer work vector structure is:

```
typedef struct P_Work_tag {
  struct {
    void *pwork_1;
    void *pwork_i;
    void *pwork_n;
  } blockName_1;
  :
  struct {
    void *pwork_1;
    void *pwork_i;
    void *pwork_n;
  } blockName_n;
  void sfunction1;
  void sfunction2;
} P_Work;
```

Blocks have an optional [vector_length] when more than one element is required for the block. This structure is created only when the number of integer work elements for all blocks in the model is greater than zero.

By default, an S-function pointer work area is defined as a single contiguous vector. However, you can change the S-function definition using the TLC function `LibDefineBlockPWork` inside the S-function's TLC file (actually, you can redefine any block's pointer work elements using this function). See the *Target Language Compiler Reference Guide* for more information about naming pointer work elements for an S-function.

## Mode Vector Structure

The mode vector structure is used to hold state information for a block function. A block should use its mode vector for a single task. That is, the block can't use part of its mode vector for one purpose and part of the mode vector for another purpose. Otherwise, the mode can't be rolled using the `%roll` directive. See the

*Target Language Compiler Reference Guide* for more details about the `%roll` directive. The structure is defined as follows:

```
typedef enum {
  DISABLED          = 0,
  ENABLED           = 1,
  BECOMING_DISABLED = 2,
  BECOMING_ENABLED  = 3
} EnableStates;

typedef Mode_tag {
  EnableStates block_1;
  EnableStates block_i;
  EnableStates block_n;
} Mode;
```

Blocks have an optional `[vector_length]` when more than one element is required for the block. This structure is created only when the number of mode elements in the model is greater than 0.

## External Inputs Structure

The external inputs structure consists of all root-level Inport block signals. Field names are determined by either the block's output signal name, when present, or by the inport block's name when the output signal is left unlabeled.

```
typedef struct ExternalInputs_tag {
  real_T BlockOutputSignal_1;
  real_T BlockOutputSignal_j;
  real_T BlockOutputSignal_n;
} ExternalInputs;
```

The root-level Inport block output signals have an optional `[vector_length]` for wide signals. The structure is created only if there are root-level inport blocks in your model.

## External Outputs Structure

The external outputs structure consists of all Outport blocks. Field names are determined by the root-level Outport block names in your model.

```
typedef struct ExternalInputs_tag {
  real_T OutportBlockName_1;
  real_T OutportBlockName_j;
  real_T OutportBlockName_n;
} ExternalOutputs;
```

The root-level Outport block output signals have an optional `[vector_length]` for wide signals. The external outputs structure is created only if there are root-level Outport blocks in your model.

## Previous Zero-Crossings Structure

Zero crossing events refer to when a particular signal, generally the input signal to a block, crosses some threshold. To determine if a signal has crossed a threshold, the previous value of the signal is saved. These signals are saved in the previous zero-crossings structure.

```
typedef enum {
  NEG_ZCSIG         = -1,
  ZERO_ZCSIG        = 0,
  POS_ZCSIG         = 1,
  UNITIALIZED_ZCSIG = INT_MAX
} ZCSigState;

typedef struct PrevZCSigState_tag {
  ZCSigState block_1;
  ZCSigState block_i;
  ZCSigState block_n;
} PrevZCSigState;
```

Blocks will have an optional `[vector_length]` for wide signals. This structure is created only when your model contains blocks that keep track of zero crossing events.

## Data Store Memory Structure

The data store memory structure is used for memory storage of the data stores in your model. The general form of the data store memory structure is:

```
typedef struct DataStore_tag {
  real_T dataStoreIdentifier1;
  :
  real_T dataStoreIdentifiern;
} DataStore;
```

Data stores have an optional [vector_length] for wide signals. This structure is created only when your model contains data store blocks.

## model.prm

The *model*.prm file contains all structure declarations. This file is included once at the top of *model*.c. The file *model*.prm is separated from *model*.c to improve readability. The general format of the *model*.prm is:

```
/* Default values for the modifiable (run-time) Parameters: */

static Parameters rtP = {
  {
    /* Block Type: Block Name */
    {
      0.240000000000000E+00,     /* parameter 1 */
      :
      0.975433948590388E-03      /* parameter i */
    }

    /* Block Type: Block Name */
    {
      0.240000000000000E+00,     /* parameter1[0] */
      0.240000000000000E+00,     /* parameter1[1] */
      0.240000000000000E+00,     /* parameter1[2] */
      0.975433948590388E-03      /* parameter2    */
    }
    :
}

/* Block I/O Structure */
BlockIO rtB;

/* States Structure */
States rtX;

/* External Outputs Structure */
ExternalOutputs rtY;

/* Parent Simstruct */
SimStruct model_S;
SimStruct *const rtS = &model_S;
```

Note that parameters are specified using C format `%#.16E`, which is the default TLC number replacement. Also, fields containing `rtInf`, `rtMinusInf`, and `rtNaN` are initially set to zero, and then reset to their appropriate numbers in the registration function.

## model.reg

The *model*.reg file contains the model registration function, MdlInitializeSizes and MdlInitialSampleTimes, which are responsible for initializing the SimStruct. These functions need to be called before the MdlStart routine is called. The name of the model registration function is always the name of your model. The general form of *model*.reg is:

```
void MdlInitializeSizes(void)
{
  (Set size of the various data structures in the SimStruct)
}

void MdlInitializeSampleTimes(void)
{
 (Set sample times for the model in the SimStruct)
}

void model(void)
{
 (Set up working areas in the SimStruct)
  (Set up any non-inlined S-functions)

}
```

## void MdlInitializeSizes(void)

The `MdlInitializeSizes` function is responsible for setting sizes of the various data structures in the `SimStruct`. The general form of the `MdlInitializeSizes` routine is:

```
void MdlInitializeSizes(void)
{
  ssSetNumContStates(rtS, NumContStates);
  ssSetNumDiscStates(rtS, NumDiscStates);
  ssSetNumOutputs(rtS, NumModelOutputs);
  ssSetNumInputs(rtS, NumModelInputs);
  ssSetDirectFeedThrough(rtS, DirectFeedThrough);
  ssSetNumSampleTimes(rtS, NumSampleTimes);
  ssSetNumRWork(rtS, NumRWork);
  ssSetNumIWork(rtS, NumIWork);
  ssSetNumPWork(rtS, NumPWork);
  ssSetNumModes(rtS, NumModes);
  ssSetNumBlocks(rtS, NumBlocks);
  ssSetNumBlockIO(rtS, NumBlockOutputs);
  ssSetNumBlockParams(rtS, NumBlockParams);
}
```

The various data structures are defined in the *model*`.h` header file. The value for each size is inlined into the source code.

## void MdlInitializeSampleTimes(void)

The `MdlInitializeSampleTimes` function is responsible for registering the sample times of your model in the `SimStruct`. The general contents of this routine for a model consisting of `N` sample times is:

```
ssSetSampleTime(rtS, 0, 0.1);
ssSetOffsetTime(rtS, 0, 0.0);
    :
ssSetSampleTime(rtS, N-1, value)
ssSetOffsetTime(rtS, N-1, value)
```

## File Splitting

To support compilers with file size limitations, the source code is split whenever the code size exceeds a specified threshold. This is controlled using the TLC variable `FileSizeThreshold`, which by default is set to 50,000 lines. To illustrate how the source code splits, assume the model's output section is being generated when the file size threshold is exceeded:

Inside *model*`.c`:

```
void MdlOutputs(int_T tid) {
  :
  :
  /* File size threshold exceeded. Splitting outputs into
     source file model1.c */
  {
      extern void MdlOutputs_split1(int_T tid);
      MdlOutputs_split1(tid);
  }
}
/* EOF model.c */
```

Inside *model1*`.c`

```
void MdlOutputs_split1(int_T tid) {
  :
  : continues where model.c left off
  :
}
:
:
```

The file *model1*`.c` includes *model*`.h`, but not *model*`.prm`.

# Program Architecture

## Introduction

The Real-Time Workshop, in addition to generating code for Simulink models, provides a framework for implementing real-time application programs. This chapter describes the framework and the structure of programs created with it.

This chapter provides information on:

- Real-time program architecture
- Source code modules for real-time programs
- Model code execution mechanism
- Contents of the `SimStruct` data structure
- Real-time program execution in single and multitasking environments
- Creating models that use multiple sample times

## Program Framework

Generating code for a Simulink model results in four files — *model*.c, *model*.h, *model*.prm, and *model*.reg, where *model* is the name of the Simulink model. This code implements the model's system equations, contains block parameters, and performs initialization.

The Real-Time Workshop's program framework provides the additional source code necessary to build the model code into a complete, stand-alone program. The program framework consists of *application modules* (files containing source code to implement required functions) designed for a number of different programming environments.

The automatic program builder ensures the program is created with the proper modules once you have configured your template makefile.

### The Common API

The application modules and the code generated for a Simulink model are implemented using a common API (application program interface). This API defines a data structure (called a `SimStruct`) that encapsulates all data for your model. The model code and the common API are described in detail in Chapter 5, "Model Code,"

This API is similar to that of S-functions, with one major exception: the API assumes that there is only one instance of the model, whereas S-functions can have multiple instances. The function prototypes also differ from S-functions.

## Real-Time Program Architecture

The structure of a real-time program consists of three *components*. Each component has a dependency on a different part of the environment in which the program executes. The following diagram illustrates this structure.

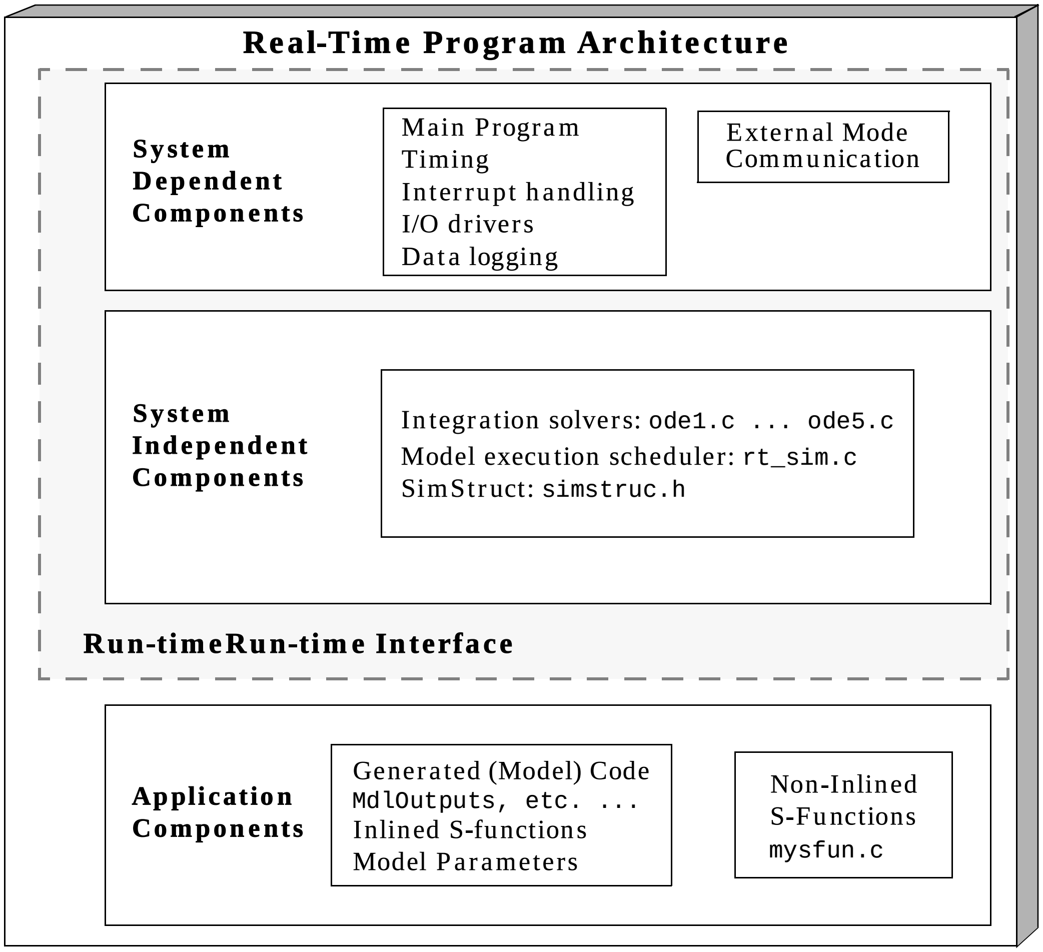

Real-Time Program Architecture
System Dependent Components
Main Program
Timing
Interrupt handling
I/O drivers
Data logging
External Mode Communication
System Independent Components
Integration solvers: ode1.c ... ode5.c
Model execution scheduler: rt_sim.c
SimStruct: simstruc.h
Run-timeRun-time Interface
Application Components
Generated (Model) Code
MdlOutputs, etc. ...
Inlined S-functions
Model Parameters
Non-Inlined S-Functions
mysfun.c

**Figure 6-1: The Program Architecture of the Real-Time Workshop**

The Real-Time Workshop architecture consists of three parts. The first two components, system dependent and independent, together form the run-time interface.

This architecture readily adapts to a wide variety of environments by isolating the dependencies of each program component. The following sections discuss each component in more detail and include descriptions of the application modules that implement the functions carried out by the system dependent, system independent, and application components.

The system dependent and independent components can be viewed together collectively as the *run-time interface*. The run-time interface is responsible for managing the execution of the real-time program created from your Simulink block diagram. The object-oriented view of the real-time program, at the highest level, is:

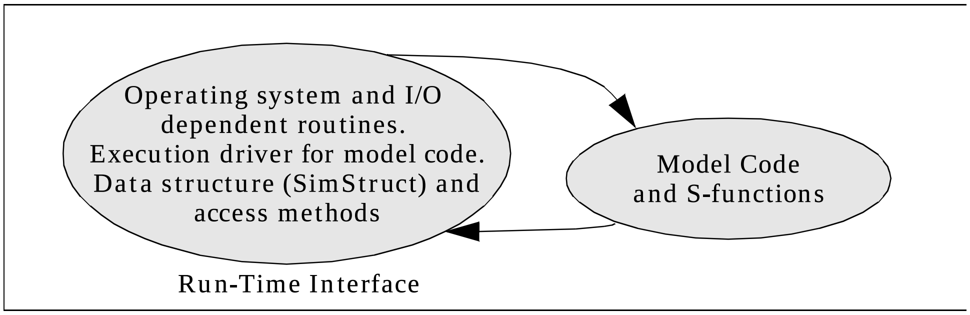
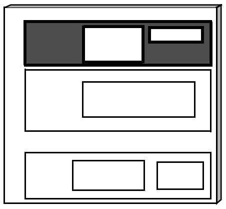

Operating system and I/O dependent routines.
Execution driver for model code.
Data structure (SimStruct) and access methods
Model Code and S-functions
Run-Time Interface

**Figure 6-2: The Object-Oriented View of a Real-time Program**

## System Dependent Components



These components contain the program's main function, which controls program timing, creates tasks, installs interrupt handlers, enables data logging, and performs error checking.

The way in which application modules implement these operations depends on the type of computer. This means that, for example, the components used for a DOS-based program perform the same operations, but differ in method of implementation from components designed to run under Tornado on a VME target.

### The `main` Function

The `main` function in a C program is the point where execution begins. In Real-Time Workshop application programs, the `main` function must perform certain operations. These operations can be grouped into three categories — initialization, model execution, and program termination.

#### Initialization

- Initialize special numeric parameters: `rtInf`, `rtMinusInf`, and `rtNaN`. These are variables that the model code can use.
- Call the model registration function to get a pointer to the `SimStruct`. The model registration function has the same name as your model. It is responsible for initializing `SimStruct` fields and any S-functions in your model.
- Initialize the model size information in the `SimStruct`. This is done by calling `MdlInitializeSizes`.
- Initialize a vector of sample times and offsets (for systems with multiple sample rates). This is done by calling `MdlInitializeSampleTimes`.
- Get the model ready for execution by calling `MdlStart`, which initializes states and similar items.
- Set up the timer to control execution of the model.
- Define background tasks and enable data logging, if selected.

### Model Execution

- Execute a background task, for example, communicate with the host during external mode simulation or introduce a wait state until the next sample interval.
- Execute model (initiated by interrupt).
- Log data to buffer (if data logging is used).
- Return from interrupt.

### Program Termination

- Call a function to terminate the program if it is designed to run for a finite time — destroy the `SimStruct`, deallocate memory, and write data to a file.

These operations are performed with a program of the form:

```
call initialization routine
install interrupt service routine

while (time < final time)
{
  background task
}
reset interrupts

complete background task
call program termination routine
```

At the interval specified by the program's base sample rate, the ISR preempts the background task to execute the model code. The base sample rate is the fastest rate in the model. If the model has continuous blocks, then the integration step size determines the base sample rate.

For example, if the model code is a controller operating at 100 Hz, then every 0.01 seconds the background task is interrupted, the controller reads its inputs from the ADC, calculates its outputs, writes these outputs to the DAC, and updates its states. Program control then returns to the background task. All of these steps must occur before the next interrupt.

## Program Timing

Real-time programs require careful timing of interrupts to ensure that the model code executes to completion before another interrupt occurs. This includes time to read and write data to and from external hardware.

The following diagram illustrates interrupt timing.

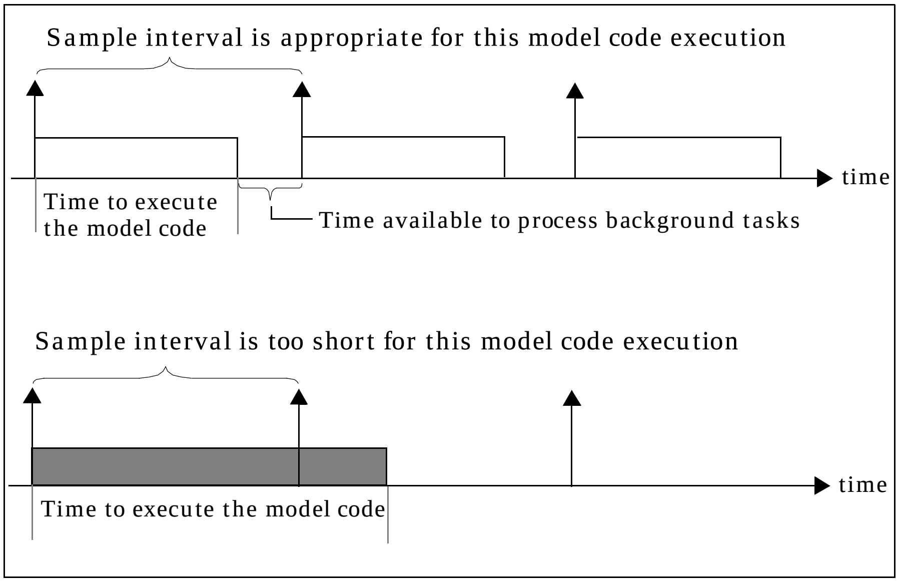

Sample interval is appropriate for this model code execution
time
Time to execute the model code
Time available to process background tasks
Sample interval is too short for this model code execution
time
Time to execute the model code

**Figure 6-3: Interrupt Timing**

The sample interval must be long enough to allow model code execution between interrupts.

In the figure above, the time between two adjacent vertical arrows is the sample interval. The empty boxes in the upper diagram show an example of a program that can complete one step within the interval and still allow time for the background task. The cross-hatched box in the lower diagram indicates what happens if the sample interval is too short. Another interrupt occurs before the task is complete. Such timing results in an execution error.

Note also that, if the real-time program is designed to run forever (i.e., the final time is infinite so the while loop never exits), then the routines that reset the interrupts, write data to a file, and destroy the `Simstruct` never execute.

## Program Execution

As the previous section indicates, a real-time program may not require 100% of the CPU's time. This provides an opportunity to run background tasks during the free time.

Background tasks include operations like writing data to a buffer or file, allowing access to program data by third-party data monitoring tools, or using Simulink external mode to update program parameters.

It is important, however, that the program be able to preempt the background task at the appropriate time to ensure real-time execution of the model code.

The way the program manages tasks depends on capabilities of the environment in which it operates.

## External Mode Communication

External mode allows communication between the Simulink block diagram and the stand-alone program that is built from the generated code. In this mode, the real-time program functions as an interprocess communication server, responding to requests from Simulink. See Chapter 4, "External Mode, Data Logging, and Signal Monitoring," for information on external mode.

## Data Logging

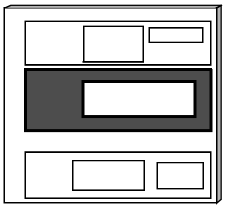
You can use the built-in data logging capabilities provided by the Real-Time Workshop or you can use third party data logging tools with your program (see Chapter 4, **"**External Mode, Parameter Tuning, and Data Logging" for an example).

The MathWorks also provides data logging facilities for creating a *model*`.mat` file at the completion of the model execution. This data logging facility is useful for code validation and high-speed simulations. See the generic real-time target, `grt`, described in "Building Generic Real-Time Programs" on page 2-7.

## Application Modules for System Dependent Components

The application modules contained in the system dependent components generally include a main module such as `rt_main.c` containing the main entry point for C. There may also be additional application modules for such things as I/O support and timer handling.

## System Independent Components



These components are collectively called system independent because all environments use the same application modules to implement these operations. This section steps through the model code (and if the model has continuous states, calls one of the numerical integration routines). This section also includes the code that defines, creates, and destroys the Simulink data structure (`SimStruct`). The model code and all S-functions included in the program define their own `SimStruct`.

The model code execution driver calls the functions in the model code to compute the model outputs, update the discrete states, integrate the continuous states (if applicable), and update time. These functions then write their calculated data to the `SimStruct`.

### Model Execution

At each sample interval, the main program passes control to the model execution function, which executes one step though the model. This step reads inputs from the external hardware, calculates the model outputs, writes outputs to the external hardware, and then updates the states.

The following diagram illustrates these steps:

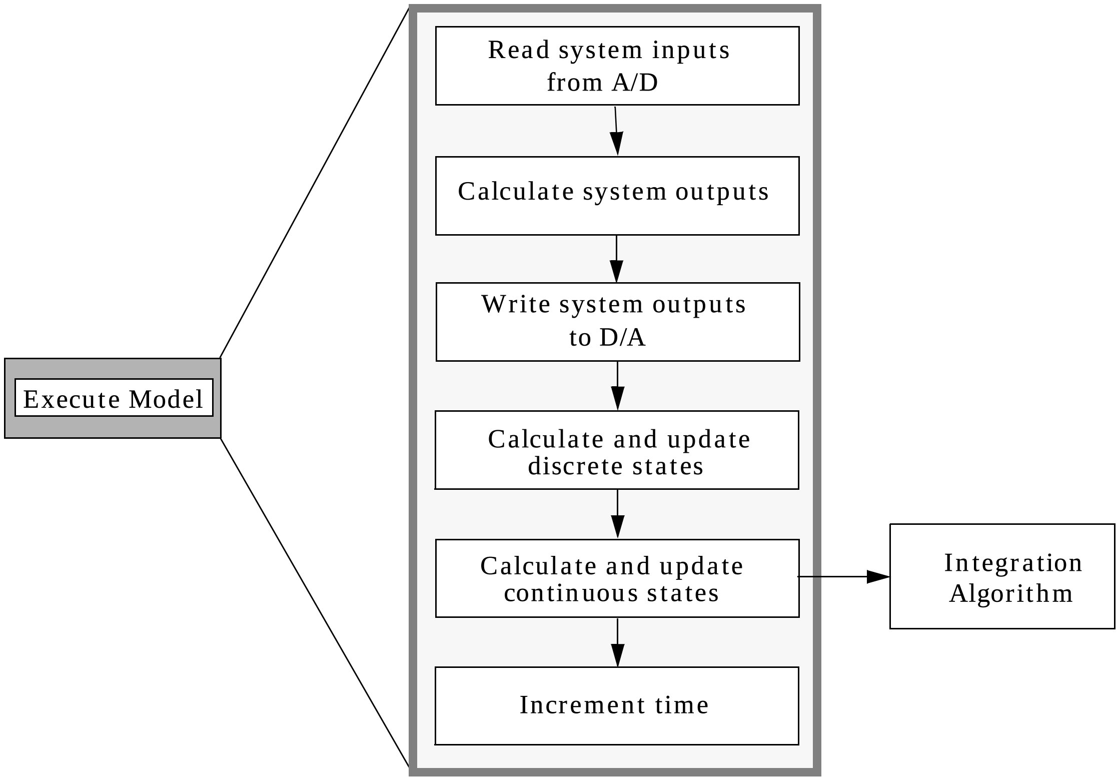

Execute Model
Read system inputs from A/D
Calculate system outputs
Write system outputs to D/A
Calculate and update discrete states
Calculate and update continuous states
Increment time
Integration Algorithm

**Figure 6-4: Executing the Model**

Note that this scheme writes the system outputs to the hardware before the states are updated. Separating the state update from the output calculation minimizes the time between the input and output operations.

## Integration of Continuous States

The real-time program calculates the next values for the continuous states based on the derivative vector, $dx/dt$, for the current values of the inputs and the state vector.

These derivatives are then used to calculate the next value of the states using a state-update equation. The state-update equation for the first order Euler method (`ode1`) is simply:

$$x = x + \frac{dx}{dt}h$$

where $h$ is the step size of the simulation, $x$ represents the state vector, and $dx/dt$ is the vector of derivatives. Other algorithms may make several calls to the output and derivative routines to produce more accurate estimates.

Note, however, that real-time programs use a fixed-step size since it is necessary to guarantee the completion of all tasks within a given amount of time. This means that, while you should use higher order integration methods for models with widely varying dynamics, the higher order methods require additional computation time. In turn, the additional computation time may force you to use a larger step size, which can diminish the accuracy increase initially sought from the higher order integration method.

Generally, the stiffer the equations, (i.e., the more dynamics in the system with widely varying time constants), the higher the order of the method that you must use.

In practice, the simulation of very stiff equations is impractical for real-time purposes except at very low sample rates. You should test fixed-step size integration in Simulink to check stability and accuracy before implementing the model for use in real-time programs.

For linear systems, it is more practical to convert the model that you are simulating to a discrete time version, for instance, using the `c2d` function in the Control System Toolbox.

## Application Modules for System Independent Components

The system independent components include these modules:

- `rt_sim.c` — Performs the activities necessary for one time step of the model. It calls the model function to calculate system outputs and then updates the discrete and continuous states.
- `ode1.c, ode2.c, ode3.c, ode4.c, ode5.c` — These modules implement the integration algorithms supported for real-time applications. See the

Simulink documentation for more information about these fixed-step solvers.

- `simstruc.h` — Contains actual definition of the Simulink data structure and the definition of the `SimStruct` access macros.

The system independent components also include code that defines, creates, and destroys the Simulink data structure (`SimStruct`). The model code and all S-functions included in the program define their own `SimStruct`.

The `simstruct` data structure encapsulates all the data relating to the model or S-function, including block parameters and outputs. Each `SimStruct` contains the following fields:

- `modelName`
- `path`
- `parent`
- `root`
- `status`
- `sizes`
- `sfcnParams`
- `states`
- `work`
- `mdlInfo`
- `callsys`
- `stInfo`

This table lists the data contained in each field of the `SimStruct`. Additional information reserved for use by Simulink and the run-time interface is provided in the `Simstruct`:

**Table 5-1: Fields in the SimStruct Data Structure**

| Field | Data Contained in Field |
|---|---|
| `modelName` | Name of the Simulink model (`ssGetModelName`) |
| `path` | Full “Simulink path” to this model (`ssGetPath`) |
| `parent` | Parent SimStruct (`ssGetParentSS`) |

**Table 5-1: Fields in the SimStruct Data Structure (Continued)**

| Field | Data Contained in Field |
|---|---|
| `root` | Root SimStruct (`ssGetRootSS`) |
| `status` | Used by S-function blocks to report error status. |
| `sizes` | Size information about the model (e.g., number of states, number of input and outputs, number of sample times). The most common size access methods are: <br>• `ssGetNumContStates, ssSetNumContStates` — Get and set the number of continuous states. <br>• `ssGetNumDiscStates, ssSetNumDiscStates` — Get and set the number of discrete states. <br>• `ssGetNumTotalStates` — Get the number of continuous and discrete states. <br>• `ssGetNumOutputs, ssSetNumOutputs` — Get and set the number of outputs. For S-functions, this is the width of the S-function block output port. For the model code, you can get, but not set the width of the output vector, which is defined by the number of root-level Outport blocks in your block diagram. <br>• `ssGetNumInputs, ssSetNumInputs`, — Get and set the number of inputs. For S-functions, this is the width of the S-function block input port. For the model code, you can get, but not set the width of the input vector, which is defined by the number of root-level Inport blocks in your block diagram. |

**Table 5-1: Fields in the SimStruct Data Structure (Continued)**

| Field | Data Contained in Field |
|---|---|
| `sizes` (continued) | • `ssGetNumRWork, ssSetNumRWork` — Get and set the number of real work elements. Real work elements are working areas for S-functions and the model code. For the model code, you can get, but not set the number of real work elements.<br>• `ssGetNumIWork, ssSetNumIWork` — Get and set the number of integer work elements. Integer work elements are working areas for S-functions and the model code. For the model code, you can get, but not set the number of integer work elements.<br>• `ssGetNumPWork, ssSetNumPWork` — Get and set the number of pointer work elements. Pointer work elements are working areas for S-functions and the model code. For the model code, you can get, but not set the number of pointer work elements.<br>• `ssGetNumSampleTimes, ssSetNumSampleTimes` — Get and set the number of sample times in your S-function or model. For the model code, you can get but not set the number of sample times in your model. |
| `sfcnParams` | S-function parameters passed in to your S-function block (see the Parameters: field of the your **S-Function** dialog box). The most common access method for this field is:<br>• `ssGetSFcnParam` — Get the S-function parameter for use in your S-function code. |

**Table 5-1: Fields in the SimStruct Data Structure (Continued)**

| Field | Data Contained in Field |
|---|---|
| `states` | Contains various vectors (e.g., block inputs and outputs, block parameter vector, work vectors). The more common access methods for this field are:<br>• `ssGetU` — Get the input vector to your S-function. For models this returns a pointer to the external inputs of your block diagram.<br>• `ssGetY` — Get the output vector of your S-function. For models this returns a pointer to the external outputs of your block diagram.<br>• `ssGetX` — Get the state (continuous followed by discrete) vector of your S-function. For models this returns a pointer to the states of your block diagram.<br>• `ssGetdX` — Get the derivative vector of your S-function. For models this returns a pointer to the derivatives of your block diagram. |
| `work` | Various work areas (`rwork`, `iwork`, `pwork`, user data, etc.). The more common access methods for this field are:<br>• `ssGetPWork` — Get the real work vector for your S-function. For models this returns a pointer to all real work elements in your block diagram.<br>• `ssGetIWork` — Get the integer work vector for your S-function. For models this returns a pointer to all integer work elements in your block diagram.<br>• `ssGetRWork` — Get the pointer work vector for your S-function. For models this returns a pointer to all pointer work elements in your block diagram. |

**Table 5-1: Fields in the SimStruct Data Structure (Continued)**

| Field | Data Contained in Field |
|---|---|
| `mdlInfo` | Model-wide information. Each S-function in your model has it's own `SimStruct`; however, there is only one `mdlInfo` field and all `SimStruct` including the root `SimStruct` for your block diagram contain a pointer to this field. The more common access methods of this field are:<br><br>• `ssGetT` — Get the current time. Note that S-functions should not use this directly, unless they have a single continuous sample time. Rather S-functions should use the `ssGetTaskTime` access method.<br><br>• `ssGetTaskTime` — Get the current task time. For S-functions, the calling convention is generally `ssGetTaskTime(S,tid)`.<br><br>• `ssGetTStart` — Get the execution start time. For most models, especially real-time systems, this is generally 0. This is defined in the **Simulation parameters** dialog box.<br><br>• `ssGetTFinal` — Get the final or stop time of your model. This is defined in the **Simulation parameters** dialog box.<br><br>• `ssIsMinorTimeStep, ssIsMajorTimeStep` — Used to determine what type of time step your model is in. The solvers execute on minor time steps; the major time steps are when blocks actually execute for a time hit. |

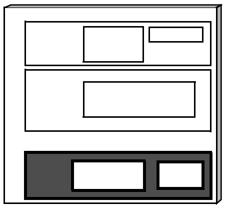
**Table 5-1: Fields in the SimStruct Data Structure (Continued)**

| Field | Data Contained in Field |
|---|---|
| `callSys` | This field is used for function-call subsystems. The more common access methods are:<br>• `ssSetCallSystemOutput(S,element)` — Indicate which output element of your S-function execute function-call subsystems.<br>• `ssCallSystem(S,element)` — Execute a function-call subsystem. |
| `stInfo` | Sample and offset times. The more common access methods are:<br>• `ssSetSampleTime(S, sample-time-index, value)` — Set the sample time.<br>• `ssSetOffsetTime(S, sample-time-index, value)` — Set the offset of the sample time. |

Access to this data structure is accomplished strictly through the use of access functions defined by the `SimStruct` API.

These functions provide access to each field in the `SimStruct`. They are used by all application modules in the real-time framework. See the *Using Simulink* manual for more information on S-functions.

## Application Components



The application components contain the generated code for the Simulink model, including the code for any S-functions in the model. This code is referred to as the model code because these functions implement the Simulink model.

However, the generated code contains more than just functions to execute the model (as described in the previous section). There are also functions to perform initialization, facilitate data access, and complete tasks before program termination. To perform these operations, the generated code must define functions that:

- Create the `SimStruct`.
- Initialize model size information in the `SimStruct`.
- Initialize a vector of sample times and sample time offsets and store this vector in the `SimStruct`.
- Store the values of the block initial conditions and program parameters in the `SimStruct`.
- Compute the block and system outputs.
- Update the discrete state vector.
- Compute derivatives for continuous models.
- Perform an orderly termination at the end of the program (when the current time equals the final time, if a final time is specified).
- Collect block and scope data for data logging (either with the Real-Time Workshop or third party tools).

### The SimStruct Data Structure

The generated code includes the file `simstruc.h`, which contains the definition of the `SimStruct` data structure. Each instance of a model (or an S-function) in the program creates its own `SimStruct`, which it uses for reading and writing data.

All functions in the generated code are public. For this reason, there can be only one instance of a model in a real-time program. This function, which always has the same name as the model, is called during program initialization to return a pointer to the `SimStruct` and initialize any S-functions.

## Model Code Functions

The functions defined by the model code are called at various stages of program execution (i.e., initialization, model execution, or program termination).

The following diagram illustrates the functions defined in the generated code and shows what part of the program executes each function.

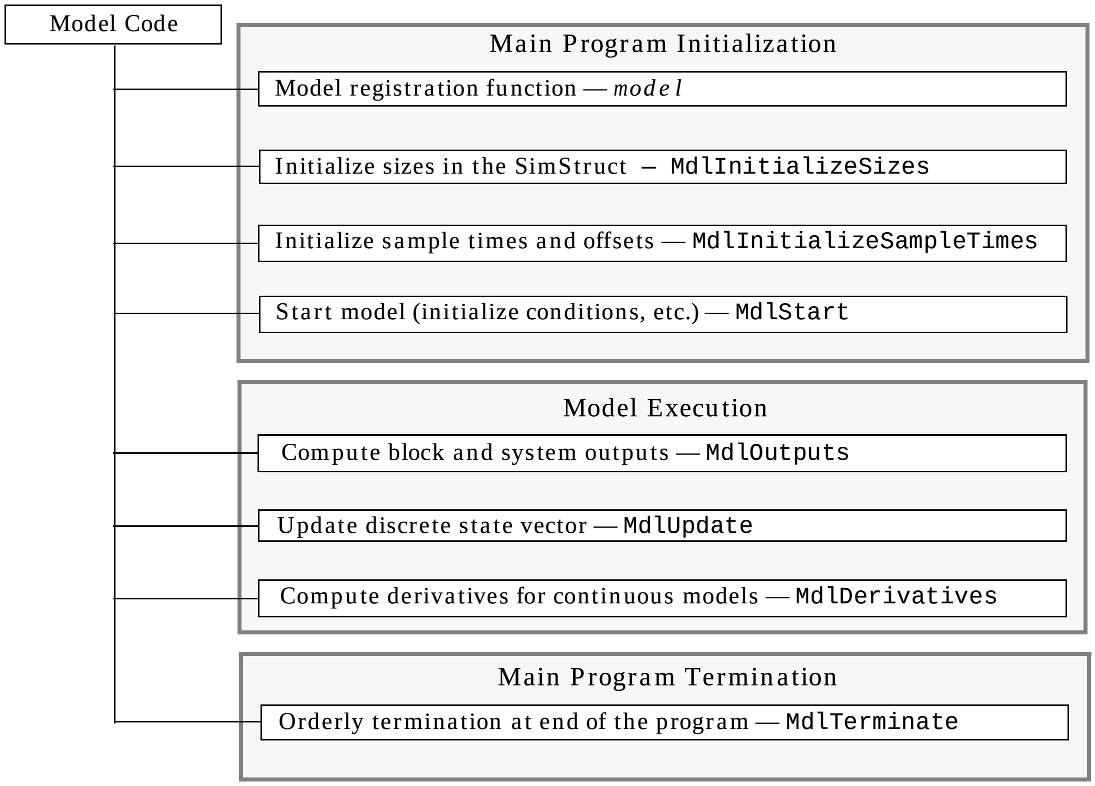

Model Code
Main Program Initialization
Model registration function — *model*
Initialize sizes in the SimStruct — `MdlInitializeSizes`
Initialize sample times and offsets — `MdlInitializeSampleTimes`
Start model (initialize conditions, etc.) — `MdlStart`
Model Execution
Compute block and system outputs — `MdlOutputs`
Update discrete state vector — `MdlUpdate`
Compute derivatives for continuous models — `MdlDerivatives`
Main Program Termination
Orderly termination at end of the program — `MdlTerminate`

**Figure 6-5: Execution of the Model Code**

This diagram shows what functions are defined in the generated code and where in the program these functions are called.

### The Model Registration Function

The model registration function has the same name as the Simulink model from which it is generated. It is called directly by the main program during initialization. Its purpose is to initialize and return a pointer to the `SimStruct`.

## Models Containing S-Functions

A *noninlined S-function* is any C MEX S-function that is not implemented using a customized `.tlc` file. If you create a C MEX S-function as part of a Simulink model, it is by default noninlined unless you write your own `.tlc` file that inlines it within the body of the *model*`.c` code. The Real-Time Workshop automatically incorporates your non-inlined C code S-functions into the program if they adhere to the S-function API described in the Simulink documentation.

This format defines functions and a `SimStruct` that are local to the S-function. This allows you to have multiple instances of the S-function in the model. The model's `SimStruct` contains a pointer to each S-function's `SimStruct`.

### Code Generation and S-Functions

If a model contains S-functions, the source code for the S-function must be on the search path the `make` utility uses to find other source files. The directories that are searched are specified in the Rules section of the template makefile that is used to build the program.

See Chapter 3, "Code Generation and the Build Process," for more information on how S-function source code is located

S-functions are implemented in a way that is directly analogous to the model code. They contain their own public registration function (which is called by the top-level model code) that initializes static function pointers in its `SimStruct`. When the top-level model needs to execute the S-function, it does so via the function pointers in the S-function's `SimStruct`. The S-functions use the same `SimStruct` data structure as the generated code; however, there can be more than one S-function with the same name in your model. This is accomplished by having function pointers to static functions.

### Inlining S-Functions

You can incorporate C MEX S-functions, along with the generated code, into the program executable. You can also write a target file for your C MEX S-function to *inline* the S-function, thus improving performance by eliminating function calls to the S-function itself. For more information on inlining S-functions, see “Inlining an S-Function,” in Chapter 3 of the *Target Language Compiler Reference Guide*.

## Application Modules for Application Components

When the Real-Time Workshop generates code, it produces four files:

- *model*`.c` — The C code generated from the Simulink block diagram. This code implements the block diagram’s system equations as well as performing initialization and updating outputs.
- *model*`.h` — Header file containing the block diagram’s simulation parameters, I/O structures, work structures, etc.
- *model*`.prm` — Header file containing parameter and other structure definitions.
- *model*`.reg` — Header file included at the bottom of `model.c`. This file contains the model registration, `MdlInitializeSizes`, and `ModelInitializeSampleTimes`.

These files are named for the Simulink model from which they are generated; *model* is replaced with the actual model name.

If you have created custom blocks using C MEX S-functions, you need the source code for these S-functions available during the build process.

# Models With Multiple Sample Rates

## Introduction

Every Simulink block can be classified according to its sample time as constant, continuous-time, discrete-time, inherited, or variable. Examples of each type include:

- Constant — Constant block, Width
- Continuous-time — Integrator, Derivative, Transfer Function
- Discrete-time — Unit Delay, Digital Filter
- Inherited — Gain, Sum, Lookup Table
- Variable — These are S-functions blocks which set their time of next hit based upon current information. These blocks work only with variable step solvers.

Blocks in the inherited category assume the sample time of the blocks that are driving them. Every Simulink block therefore has a sample time, whether it is explicit, as in the case of continuous or discrete blocks (continuous blocks have a sample time of zero), or implicit, as in the case of inherited blocks.

Simulink allows you to create models without any restrictions on the connections between blocks with disparate sample times. It is therefore possible to have blocks with differing sample times in a model (a mixed-rate system). One advantage of employing multiple sample times can be improved efficiency when executing in a multitasking real-time environment.

Simulink provides considerable flexibility in building these mixed-rate systems. However, the same flexibility also allows you to construct models for which the code generator cannot generate correct real-time code for execution in a multitasking environment. But to make these models operate correctly in real-time (i.e., give the right answers), you must modify your model. The sections that follow discuss the issues you must address to use a mixed-rate model successfully in a multitasking environment.

## Single Versus Multitasking Environments

There are two basic environments — a singletasking operating system (e.g., DOS or a *bare-board* target) and a multitasking, real-time operating system (e.g., Tornado). Note that simply being a multitasking system, such as UNIX or MS-Windows, does not guarantee that the program can execute in real-time because you cannot guarantee that the program can preempt other processes when you want it to.

In DOS, where only one process can exist at any given time, the interrupt service routine (ISR) must perform the steps of saving the processor context, executing the model code, collecting data, and restoring the processor context.

Tornado, on the other hand, provides automatic context switching and task scheduling. This simplifies the operations performed by the ISR. In this case, the ISR simply enables the model execution task, which is normally blocked.

The following diagram illustrates this difference.

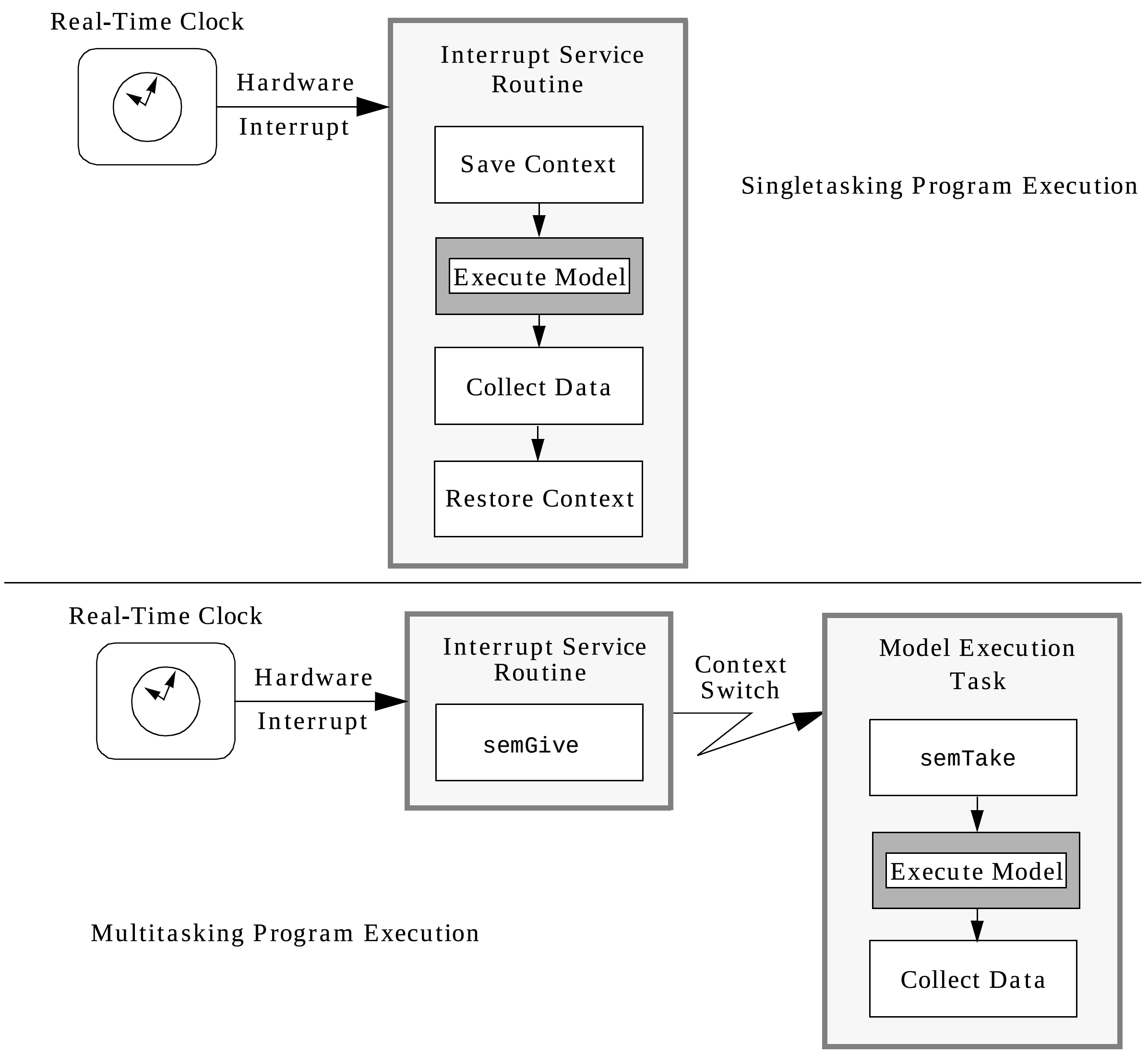


**Figure 7-1: Singletasking Versus Multitasking Program Execution**

This chapter focuses on when and how the run-time interface executes your model. See Figure 6-3 on page 6-9 for a description of what happens during model execution.

## Executing Multitasking Models

In cases where the continuous part of a model executes at a rate that is different from the discrete part, or a model has blocks with different sample rates, the code assigns each block a *task identifier* to associate it with the task that executes at its sample rate.

Certain restrictions apply to the sample rates that you can use:

- The sample rate of any block must be an integer multiple of the base (i.e., the fastest) sample rate. The base sample rate is determined by the fixed step size specified on the Solver page of the **Simulation parameters** dialog box (if a model has continuous blocks) or by the fastest sample time specified in the model (if the model is purely discrete). Continuous blocks always execute via an integration algorithm that runs at the base sample rate.
- The continuous and discrete parts of the model can execute at different rates only if the discrete part is executed at the same or a slower rate than the continuous part (and is an integer multiple of the base sample rate).

## Multitasking and Pseudomultitasking

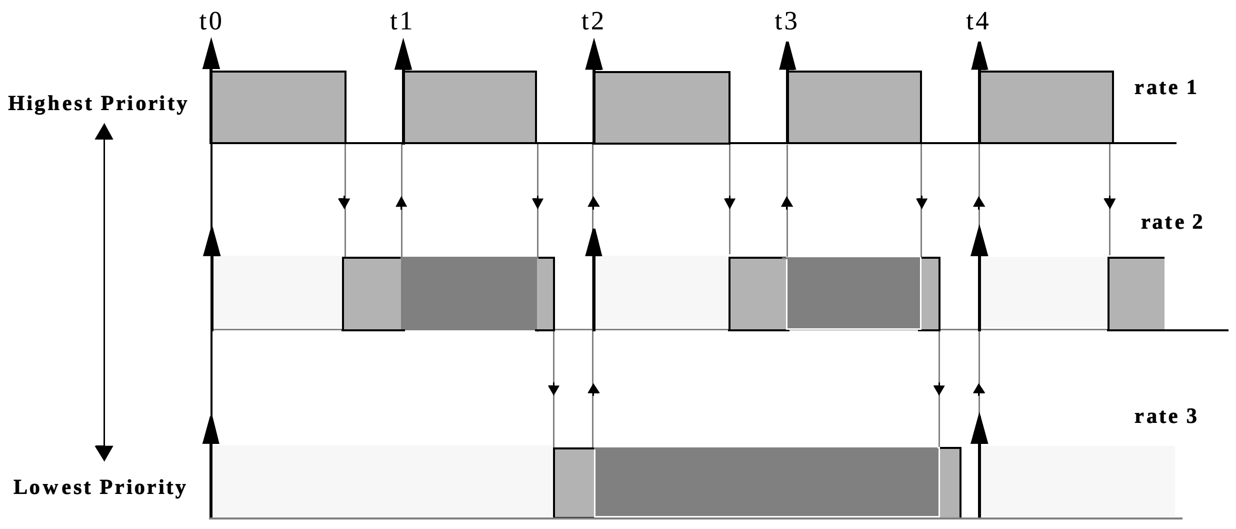
In a multitasking environment, the blocks with the fastest sample rates are executed by the task with the highest priority, the next slowest blocks are executed by a task with the next lower priority, and so on. Time available in between the processing of high priority tasks is used for processing lower priority tasks. This results in efficient program execution.

See Figure 7-2 on page 7-6 for a graphical representation of task timing.

In singletasking environments, you cannot define separate tasks and assign them priorities. However, the Real-Time Workshop application modules implement what is effectively a multitasking execution scheme using overlapped interrupts, accompanied by manual context switching.

This means an interrupt can occur while another interrupt is currently in progress. When this happens, the current interrupt is preempted, the floating-point unit (FPU) context is saved, and the higher priority interrupt executes its higher priority (i.e., faster sample rate) code. Once complete, control is returned to the preempted ISR.

The following diagrams illustrate how mixed-rate systems are handled by the Real-Time Workshop in these two environments.



↑ Vertical arrows indicate sample times.

↓ Dotted lines with downward pointing arrows indicate the release of control to a lower priority task.

▭ Dark gray areas indicate task execution.

■ Hashed areas indicate task preemption by a higher priority task.

↑ Dotted lines with upward pointing arrows indicate preemption by a higher priority task.

□ Light gray areas indicate task execution is pending.

**Figure 7-2: Multitasking System Execution**

.

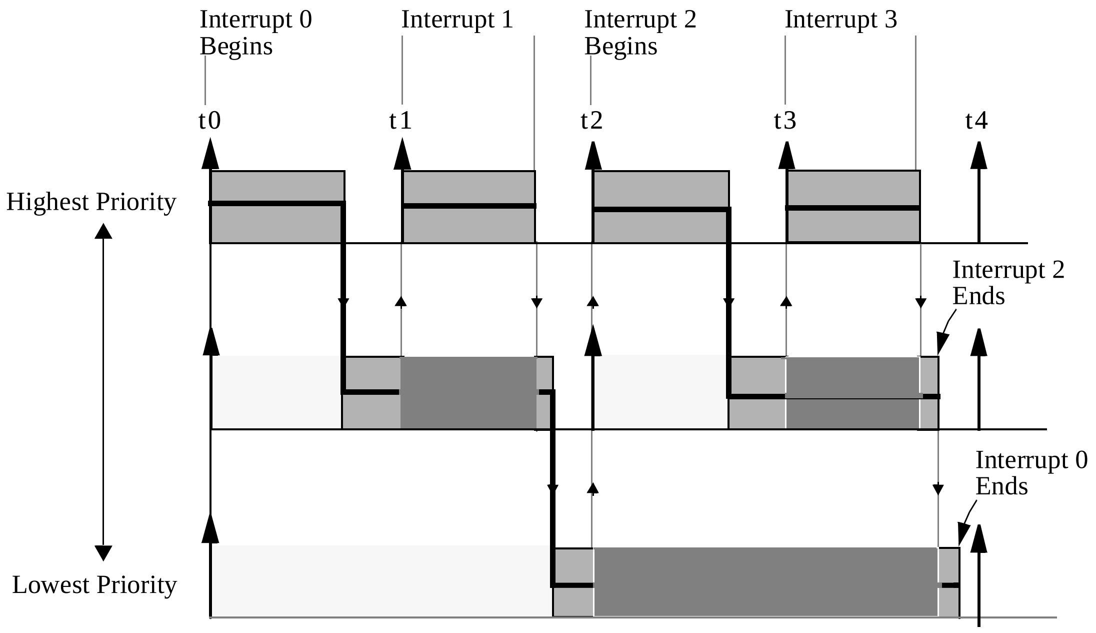


**Figure 7-3: Pseudomultitasking Using Overlapped Interrupts**

This diagram illustrates how overlapped interrupts are used to implement pseudomultitasking. Note that in this case, Interrupt 0 does not return until after Interrupts 1, 2, and 3.

## Building the Program for Multitasking Execution

To use multitasking execution, define the multitasking flag (which is defined in the template makefile). Note, by default most template makefiles are set up for singletasking. If you want to use multitasking, you can add –DMULTITASKING (or –DMT) to the compile options, usually OPTS. You can do this in the RTW page of the **Simulation parameters** dialog box, by specifying the build command as:

```
make_rtw OPTS=–DMULTITASKING
```

Alternatively, you can edit the template makefile.

The MathWorks provides support for some targets, such as DOS and Tornado. This support includes both single and multitasking environments with singletasking as the default. Refer to your real-time target documentation to determine the default tasking environment.

## Singletasking

It is possible to execute the model code in a strictly singletasking manner. While this method is less efficient with regard to execution speed, in certain situations it may allow you to simplify your model.

In a singletasking environment, the base sample rate must define a time interval that is long enough to allow the execution of all blocks within that interval.

The following diagram illustrates the inefficiency inherent in singletasking execution:

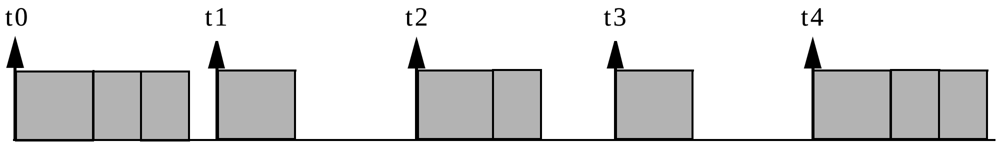


**Figure 7-4: Singletasking System Execution**

Singletasking system execution requires a sample interval that is long enough to execute one step through the entire model.

## Building the Program for Singletasking Execution

To use singletasking execution, undefine the multitasking flag (which may be defined in the template makefile). If you want to use singletasking, remove –`DMULTITASKING` (or –`DMT`) from the macro definition (if it is present). Note that by default most template makefiles are set up for singletasking.

## Model Execution

To generate code that executes correctly in real-time, you may need to modify sample rate transitions within the model before generating code. To understand this process, first consider how Simulink simulations differ from real-time programs.

## Simulating Models with Simulink

Before Simulink simulates a model, it orders all of the blocks based upon their topological dependencies. This includes expanding subsystems into the individual blocks they contain and flattening the entire model into a single list. Once this step is complete, each block is executed in order.

The key to this process is the proper ordering of blocks. Any block whose output is directly dependent on its input (i.e., any block with direct feedthrough) cannot execute until the block driving its input has executed.

Some blocks set their outputs based on values acquired in a previous time step or from initial conditions specified as a block parameter. The output of such a block is determined by a value stored in memory, which can be updated independently of its input. During simulation, all necessary computations are performed prior to advancing the variable corresponding to time. In essence, this results in all computations occurring instantaneously (i.e., no computational delay).

## Executing Models in Real-Time

A real-time program differs from a Simulink simulation in that the program must execute the model code synchronously with real-time. Every calculation results in some computational delay. This means the sample intervals cannot be shortened or lengthened (as they can be in Simulink), which leads to less efficient execution:

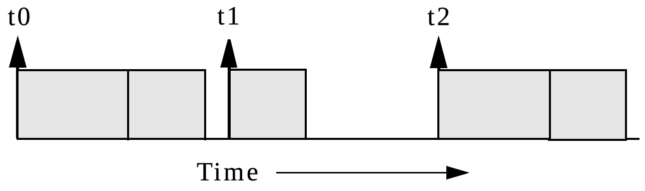


**Figure 7-5: Unused Time in Sample Interval**

Sample interval t1 cannot be compressed to increase execution speed because sample times must stay in sync with real-time.

Real-Time Workshop application programs are designed to circumvent this potential inefficiency by using a multitasking scheme. This technique defines tasks with different priorities to execute parts of the model code that have different sample rates.

See “Multitasking and Pseudomultitasking” on page 7–5 for a description of how this works. It is important to understand that section before proceeding here.

### Multitasking Operation

The use of multitasking can improve the efficiency of your program if the model is large and has many blocks executing at each rate. It can also degrade performance if your model is dominated by a single rate, and only a few blocks execute at a slower rate. In this situation, the overhead incurred in task switching can be greater than the time required to execute the slower blocks. It is more efficient to execute all blocks at the dominant rate.

If you have a model that can benefit from multitasking execution, you may need to modify your Simulink model for this scheme to generate correct results.

### Singletasking Operation

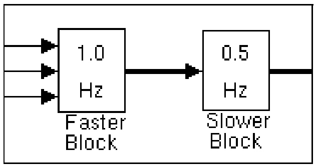
Alternatively, you can run your real-time program in singletasking mode. Singletasking programs require longer sample intervals due to the inherent inefficiency of that mode of execution.

## Sample Rate Transitions

There are two possible sample rate transitions that can exist within a model:

- A faster block driving a slower block
- A slower block driving a faster block

In singletasking systems, there are no issues involved with multiple sample rates. In multitasking and pseudomultitasking systems, however, differing sample rates can cause problems. To prevent possible errors in calculated data, you must control model execution at these transitions. In transitioning from faster to slower blocks, you must add Zero-Order Hold blocks between fast to slow transitions and set the sample rate of the Zero-Order Hold to that of the slower block:



becomes

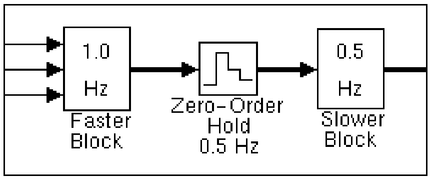
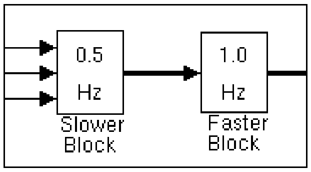


**Figure 7-6: Transitioning from Faster to Slower Blocks**

In transitioning from slower to faster blocks, you must add Unit Delay blocks between slow to fast transitions and set the sample rate of the Unit Delay to that of the slower block:



becomes

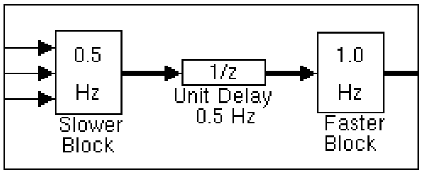
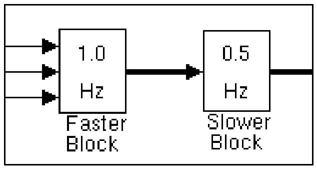


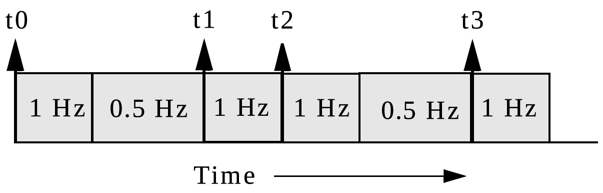
**Figure 7-7: Transitioning from Slower to Faster Blocks**

The next four sections describe the theory and reasons why Unit Delay and Zero-Order Hold blocks are necessary for sample time transitions.

## Faster to Slower Transitions in Simulink

In a model where a faster block drives a slower block having direct feedthrough, the outputs of the faster block are always computed first. In simulation intervals where the slower block does not execute, the simulation progresses more rapidly because there are fewer blocks to execute. The following diagram illustrates this situation:





Simulink does not execute in real-time, which means that it is not bound by real-time constraints. Simulink waits for, or moves ahead to, whatever tasks are necessary to complete simulation flow. The actual time interval between sample time steps can vary.

## Faster to Slower Transitions in Real-Time

In models where a faster block drives a slower block, you must compensate for the fact that execution of the slower block may span more than one execution period of the faster block. This means that the outputs of the faster block may change before the slower block has finished computing its outputs. The following diagram illustrates a situation where this problem arises. The hashed area indicates times when tasks are preempted by higher priority before completion:

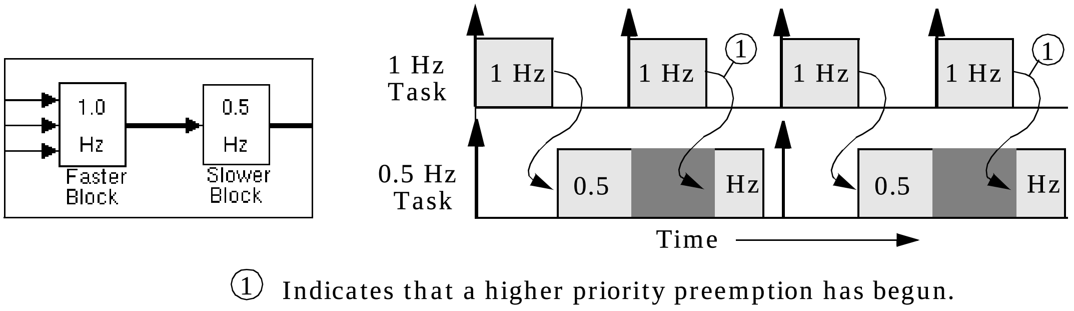
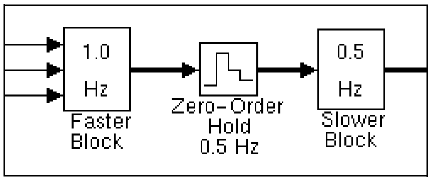


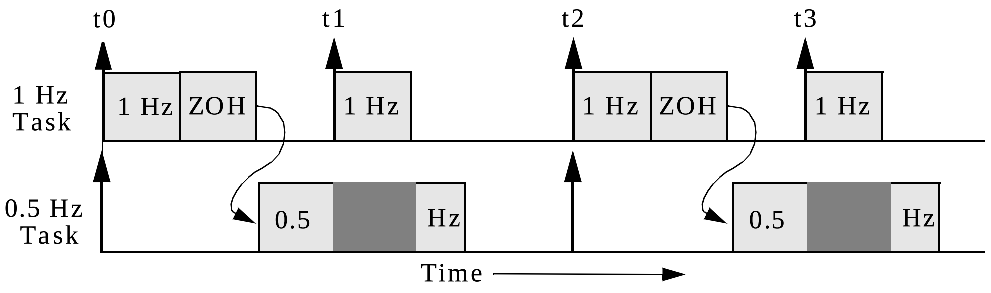
**Figure 7-8: Time overlaps in faster to slower transitions**

In this figure, the faster block executes a second time before the slower block has completed execution. This can cause unpredictable results because the input data to the slow task is changing.

To avoid this situation, you must hold the outputs of the 1 Hz (faster) block until the 0.5 Hz (slower) block finishes executing. The way to accomplish this is by inserting a Zero-Order Hold (ZOH) block between the 1 Hz and 0.5 Hz blocks. The sample time of the ZOH block must be set to 0.5 Hz (i.e., the sample time of the slower block).



The ZOH block executes at the sample rate of the slower block, but with the priority of the faster block.

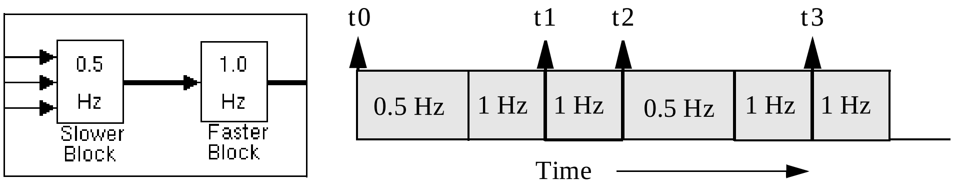


This ensures that the ZOH block executes before the 0.5 Hz block (its priority is higher) and that its output value is held constant while the 0.5 Hz block executes (it executes at the slower sample rate).

## Slower to Faster Transitions in Simulink

In a model where a slower block drives a faster block, Simulink again computes the output of the driving block first. During sample intervals where only the faster block executes, the simulation progresses more rapidly.

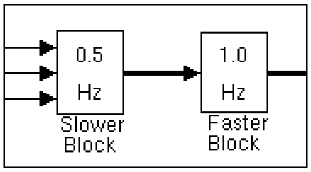
The following diagram illustrates the execution sequence:

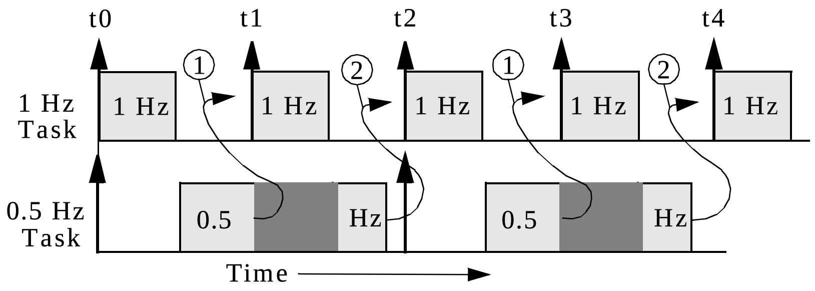


As you can see from the preceding diagrams, Simulink can simulate models with multiple sample rates in an efficient manner. However, Simulink does not operate in real-time.

## Slower to Faster Transitions in Real-Time

In models where a slower block drives a faster block, the generated code assigns the faster block a higher priority than the slower block. This means the faster block is executed before the slower block, which requires special care to avoid incorrect results.





① The faster block executes a second time prior to the completion of the slower block.

② The faster block executes before the slower block.

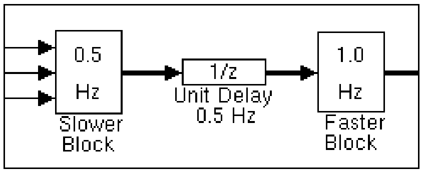
**Figure 7-9: Time Overlaps in Slower to Faster transitions**

This timing diagram illustrates two problems:

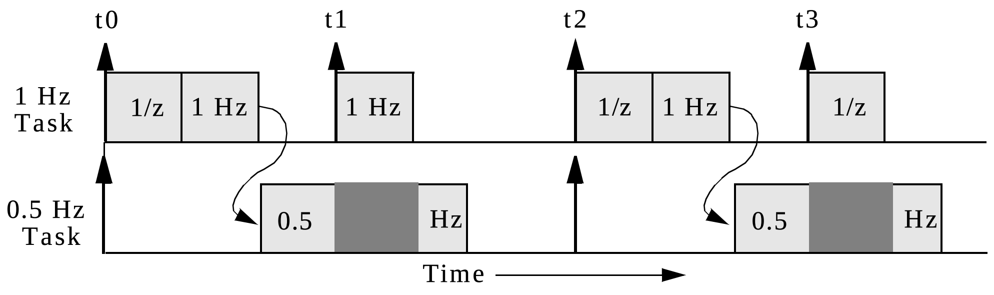
1. Execution of the slower block is split over more than one faster block interval. In this case the faster task executes a second time before the slower task has completed execution. This means the inputs to the slower task can change, causing unpredictable results.

2. The faster block executes before the slower block (which is backwards from the way Simulink operates). In this case, the 1 Hz block executes first; but the inputs to the faster task have not been computed. This can cause unpredictable results.

To eliminate these problems, you must insert a Unit Delay block between the slower and faster blocks. The sample rate for a Unit Delay block must be set to that of the block that is driving it (i.e., the slower block):



This pictures shows the timing sequence that results with the added Unit Delay block:



The output portion of a Unit Delay block is executed at the sample rate of the slower block, but with the priority of the faster block. Since a Unit Delay block drives the faster block and has effectively the same priority, it is executed before the faster block. This solves the first problem.

The second problem is alleviated because the Unit Delay block executes at a slower rate and its output does not change during the computation of the faster block it is driving.

---

**Note**: Inserting a Unit Delay block changes the model. The output of the slower block is now delayed by one time step compared to the output without a Unit Delay.

---

# Targeting Tornado for Real-Time Applications

## Introduction

This chapter describes how to create real-time programs for execution under VxWorks, which is part of the Tornado environment.

The VxWorks real-time operating system is available from Wind River Systems, Inc. It provides many UNIX-like features and comes bundled with a complete set of development tools.

**Note:** Tornado is an integrated environment consisting of VxWorks (a high-performance real-time operating system), application building tools (compiler, linker, make, and archiver utilities), and interactive development tools (editor, debugger, configuration tool, command shell, and browser).

This chapter discusses the run-time architecture of VxWorks-based real-time programs generated by the Real-Time Workshop and provides specific information on program implementation. Topics covered include:

- Configuring device driver blocks and makefile templates.
- Building the program.
- Downloading the object file to the VxWorks target.
- Executing the program on the VxWorks target.
- Using Simulink external mode to change model parameters and download them to the executing program on the VxWorks target.
- Using the StethoScope data acquisition and graphical monitoring tool, which is available as an option with VxWorks. It allows you to access the output of any block in the model (in the real-time program) and display the data on the host.

### Confirming your Tornado Setup is Operational

Before beginning, you must install and configure Tornado on your host and target hardware, as discussed in the Tornado documentation. You should then run one of the VxWorks demonstration programs to ensure you can boot your VxWorks target and download object files to it. See the *Tornado User's Guide* for additional information about installation and operation of VxWorks and Tornado products.

## Run-time Architecture Overview

In a typical VxWorks-based real-time system, the hardware consists of a UNIX or PC host running Simulink connected to a VxWorks target CPU via Ethernet. In addition, the target chassis may contain I/O boards with A/D and D/A converters to communicate with external hardware. The following diagram shows the arrangement:

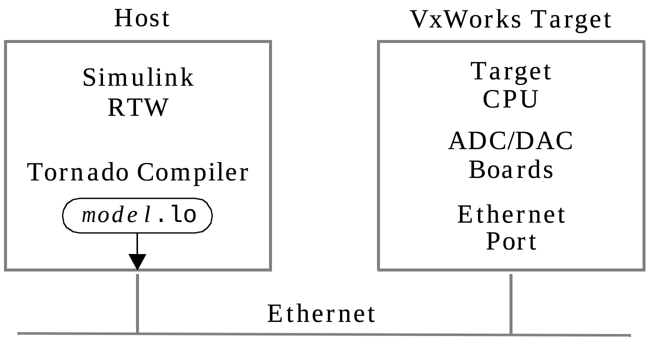


**Figure 8-1: Typical Hardware Setup for a VxWorks Application**

The real-time code is compiled on the UNIX or PC host using the cross compiler supplied with the VxWorks package. The object file (*model*.lo) output from the Real-Time Workshop program builder is downloaded, using WindSh (the command shell) in Tornado, to the VxWorks target CPU via an Ethernet connection.

The real-time program executes on the VxWorks target and interfaces with external hardware via the I/O devices installed on the target.

### Parameter Tuning and Monitoring

You can change program parameters from the host while the program executes using Simulink external mode. You can also monitor program outputs using the StethoScope data analysis tool.

Using Simulink external mode and StethoScope in combination allows you to change model parameters in your program “on the fly” and to analyze the results of these changes in real-time.

### External Mode

Simulink external mode provides a mechanism to download new parameter values to the executing program. In this mode, the external link MEX-file sends a vector of new parameter values to the real-time program via the network connection. These new parameter values are sent to the program whenever you make a parameter change without requiring a new code generation or build iteration.

The real-time program (executing on the VxWorks target) runs a low priority task that communicates with the external link MEX-file and accepts the new parameters as they are passed into the program.

Communication between Simulink and the real-time program is accomplished using the sockets network API. This implementation requires an ethernet network that supports TCP/IP. See Chapter 4, “External Mode, Data Logging, and Signal Monitoring,” for more information on external mode.

Changes to the block diagram structure (for example, adding or removing blocks) require generation of model and execution of the build process.

### Configuring VxWorks to Use Sockets

If you want to use Simulink external mode with your VxWorks program, you must configure your VxWorks kernel to support sockets by including the `INCLUDE_NET_INIT`, `INCLUDE_NET_SHOW`, and `INCLUDE_NETWORK` options in your VxWorks image. For more information on configuring your kernel, see the *VxWorks Programmer’s Guide*.

Before using external mode, you must ensure that VxWorks can properly respond to your host over the network. You can test this by using the host command

```
ping <target_name>
```

---

**Note:** You may need to enter a routing table entry into VxWorks if your host is not on the same local network (subnet) as the VxWorks system. See `routeAdd` in the *VxWorks Reference Guide* for more information.

---

### Configuring Simulink to Use Sockets

Simulink external mode uses a MEX-file to communicate with the VxWorks system. The MEX-file is:

*matlabroot*`/toolbox/rtw/ext_comm.*`

where * is a host-dependent MEX-file extension.

The Real-Time Workshop provides compiled versions of `ext_comm`, which are created from `ext_comm.c` for all supported hosts. If for any reason you want to rebuild the MEX-file, you can regenerate it by using MATLAB's `mex` command, provided that you have an installed compiler supported by the MATLAB API. See the *MATLAB Application Program Interface Guide* for more information.

This table lists the form of the commands on PC and UNIX platforms:

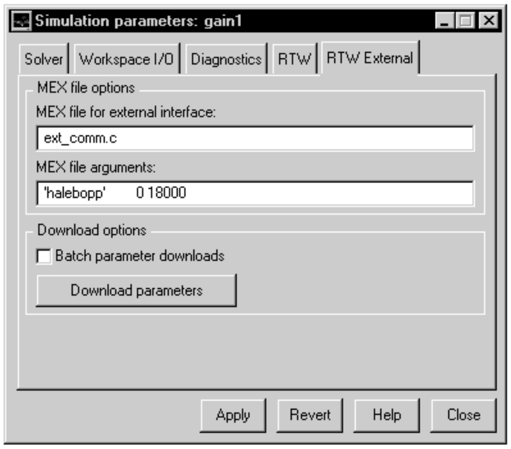
**Table 6-1: Commands Needed to Rebuild MEX-files**

| Platform | Commands |
|---|---|
| PC | `cd` *matlabroot*`\toolbox\rtw`<br>`mex` *matlabroot*`\rtw\ext_mode\ext_comm.c`<br>`    –I`*matlab*`\rtw\c\src –DWIN32`<br>`    compiler_library_path\wsock32.lib` |
| UNIX | `cd` *matlabroot*`/toolbox/rtw`<br>`mex` *matlabroot*`/rtw/ext_mode/ext_comm.c`<br>`    –I`*matlab*`/rtw/c/src` |

You then specify this file in the **RTW External** page of the **Simulation parameters** dialog box (accessed from the **Simulation** menu) as the **MEX-file for external interface**. In the **MEX-file arguments** field you must specify the name of the VxWorks target system and, optionally, the verbosity and TCP port number. Verbosity can be 0 (the default) or 1 if extra information is desired. The TCP port number ranges from 256 to 65535 (the default is 17725). If there is a conflict with other software using TCP port 17725, you can change the port that you use. The format for the MEX-file arguments field is

```
'target_network_name' [verbosity] [TCP port number]
```

This picture shows the RTW External page configured for a target system called `halebopp` with default verbosity and the port assigned to 18000:



Simulation parameters: gain1

MEX file options

MEX file for external interface: ext_comm.c

MEX file arguments: 'halebopp' 0 18000

Download options

☐ Batch parameter downloads

## StethoScope

With StethoScope, you can access the output of any block in the model (in the real-time program) and display this data on a host. Signals are installed in StethoScope by the real-time program using the `BlockIOSignals` data structure or, interactively from the `WindSh` while the real-time program is running. To use StethoScope interactively, see the *StethoScope User's Manual*.

To use StethoScope you must specify the proper options with the build command. See "Build Command Options" on page 8-15 for information on these options.

## Run-time Structure

The real-time program executes on the VxWorks target while Simulink and StethoScope execute on the same or different host workstations. Both Simulink and StethoScope require tasks on the VxWorks target to handle communication.

This diagram illustrates the structure of a VxWorks application using Simulink's external mode and StethoScope.

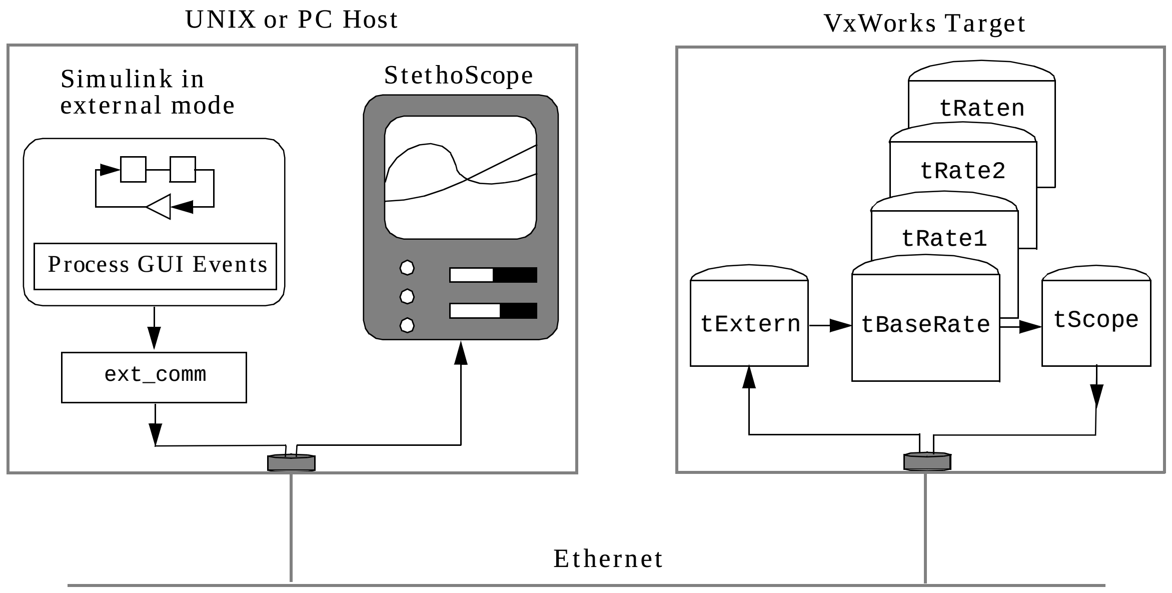


**Figure 8-2: The Run-time Structure**

The program creates VxWorks tasks to run on the real-time system: one communicates with Simulink, the others execute the model. StethoScope creates its own tasks to collect data.

### Host Processes

There are two processes running on the host side that communicate with the real-time program:

- Simulink running in external mode. Whenever you change a parameter in the block diagram, Simulink calls the external link MEX-file to download any new parameter values to the VxWorks target.
- The StethoScope user interface module. This program communicates with the StethoScope real-time module running on the VxWorks target to retrieve model data and plot time histories.

### VxWorks Tasks

There are two modes in which the real-time program can be run—singletasking and multitasking. The code for both modes is located in:

*matlabroot*`/rtw/c/tornado/rt_main.c`

The Real-Time Workshop compiles and links `rt_main.c` with the model code during the build process.

**Singletasking.** By default, the model is run as one task, `tSingleRate`. This may actually provide the best performance (highest base sample rate) depending on the model. See Chapter 7, "Models With Multiple Sample Rates," for more information on singletasking vs. multitasking.

- `tSingleRate` — Runs at the base rate of the model and executes all necessary code for the slower sample rates. Execution of the `tSingleRate` task is normally blocked by a call to the VxWorks `semTake` routine. When a clock interrupt occurs, the interrupt service routine calls the `semGive` routine, which causes the `semTake` call to return. Once enabled, the `tSingleRate` task executes the model code for one time step. The loop then waits at the top by again calling `semTake`. For more information about the `semTake` and `semGive` routines, refer to the *VxWorks Reference Manual*. By default, it runs at a relatively high priority (30), which allows it to execute without interruption from background system activity.

**Multitasking.** Optionally, the model can run as multiple tasks, one for each sample rate in the model.

- `tBaseRate` — This task executes the components of the model code run at the base (highest) sample rate. By default, it runs at a relatively high priority (30), which allows it to execute without interruption from background system activity.
- `tRate`*n* — The program also spawns a separate task for each additional sample rate in the system. These additional tasks are named `tRate1`, `tRate2`, …, `tRaten`, where `n` is slowest sample rate in the system. The priority of each additional task is one lower than its predecessor (`tRate1` has a lower priority than `tBaseRate`).

**Supporting tasks.** If you select external mode and/or StethoScope during the build process, these tasks will also be created:

- `tExtern` — This task implements the server side of a socket stream connection that accepts data transferred from Simulink to the real-time program. In this implementation, `tExtern` waits for a message to arrive from Simulink. When a message arrives, `tExtern` retrieves it and modifies the specified parameters accordingly.

  `tExtern` runs at a lower priority than `tRaten`, the lowest priority model task. The source code for `tExtern` is located in *matlabroot*`/rtw/c/src/ext_svr.c`.
- `tScopeDaemon` and `tScopeLink` — StethoScope provides its own VxWorks tasks to enable real-time data collection and display. In singletasking mode, tSingleRate collects signals; in multitasking mode, tBaseRate collects them. Both perform the collection on every base time step. The StethoScope tasks then send the data to the host for display when there is idle time, that is, when the model is waiting for the next time step to occur. `rt_main.c` starts these tasks if they are not already running.

## Implementation Overview

To implement and run a VxWorks-based real-time program using the Real-Time Workshop, you must:

- Design a Simulink model for your particular application.
- Add the appropriate device driver blocks to the Simulink model, if desired.
- Configure the `tornado.tmf` template makefile for your particular setup.
- Establish a connection between the host running Simulink and the VxWorks target via ethernet.
- Use the automatic program builder to generate the code and the custom makefile, invoke the `make` utility to compile and link the generated code, and load and activate the tasks required.

The figure below shows the Real-Time Workshop Tornado run-time interface modules and the generated code for the F-14 example from Chapter 2:

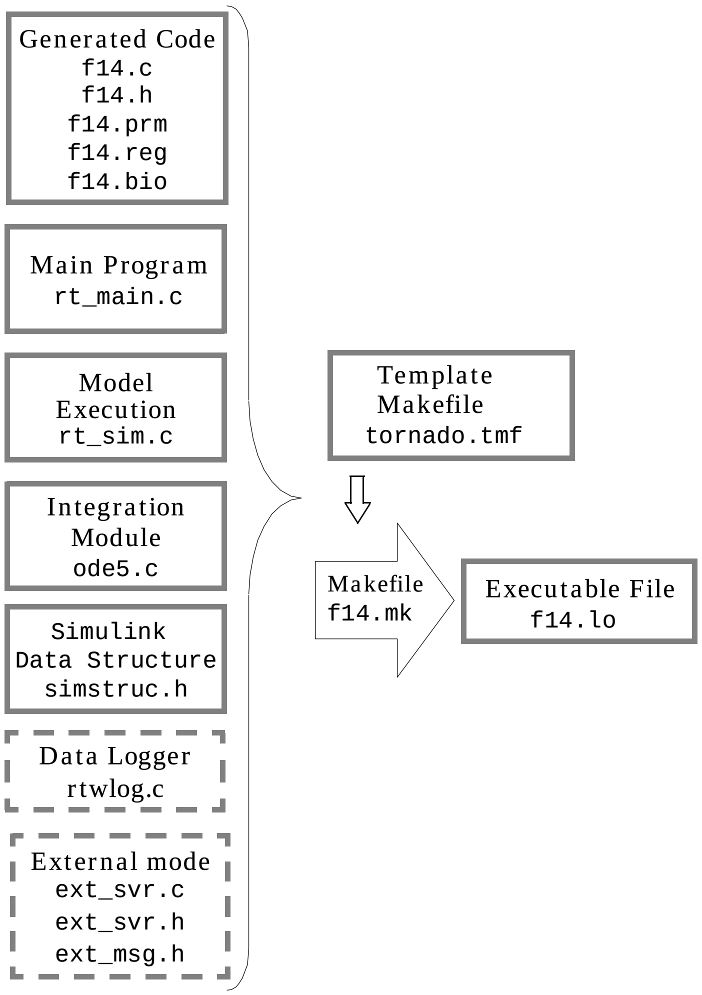


**Figure 8-3: Source Modules Used to Build the VxWorks Real-Time Program**

This diagram illustrates the code modules used to build a VxWorks real-time program. Dashed boxes indicate optional modules.

## Adding Device Driver Blocks

The real-time program communicates with the I/O devices installed in the VxWorks target chassis via a set of device drivers. These device drivers contain the necessary code that runs on the target processor for interfacing to specific I/O devices.

To make device drivers easy to use, they are implemented as Simulink S-functions using C code MEX-files. This means you can connect them to your model like any other block and the code generator automatically includes a call to the block's C code in the generated code.

You can also inline S-functions via the Target Language Compiler. Inlining allows you to restrict function calls to only those that are necessary for the S-function. This can greatly increase the efficiency of the S-function. For more information about inlining S-functions, see the *Target Language Compiler Reference Guide*.

You can have multiple instances of device driver blocks in your model. See Chapter 10, "Targeting Custom Hardware," for more information about creating device drivers.

## Configuring the Template Makefile

To configure the VxWorks template, `tornado.tmf`, you must specify information about the environment in which you are using VxWorks. This section lists the lines in the file that you must edit.

### VxWorks Configuration

To provide information used by VxWorks, you must specify the type of target and the specific CPU on the target. The target type is then used to locate the correct cross compiler and linker for your system.

The CPU type is used to define the `CPU` macro which is in turn used by many of the VxWorks header files. Refer to the *VxWorks Programmer's Guide* for information on the appropriate values to use.

This information is in the section labeled:

```
#-------------- VxWorks Configuration --------------
```

Edit the following lines to reflect your setup:

```
VX_TARGET_TYPE = 68k
CPU_TYPE = MC68040
```

### Downloading Configuration

In order to perform automatic downloading during the build process, the target name and host name that the Tornado target server will run on must be specified. Modify these macros to reflect your setup.

```
#-------------- Macros for Downloading to Target--------------
TARGET = targetname
TGTSVR_HOST = hostname
```

## Tool Locations

In order to locate the Tornado tools used in the build process, the following three macros must either be defined in the environment or specified in the template makefile. Modify these macros to reflect your setup.

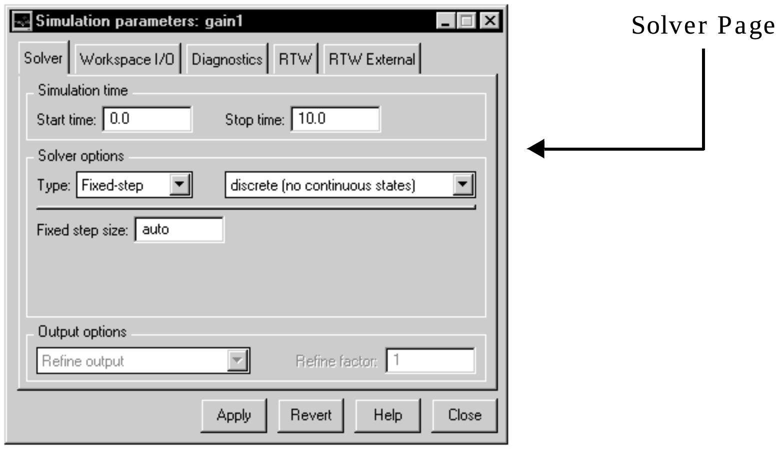
```
#-------------- Tool Locations --------------
WIND_BASE = c:/Tornado
WIND_HOST_TYPE = x86-win32
WIND_REGISTRY = $(COMPUTERNAME)
```

## Building the Program

Once you have created the Simulink block diagram, added the device drivers, and configured the makefile template, you are ready to set the build options and initiate the build process.

## Specifying the Real-Time Build Options

Set the real-time build options using the **Solver** and **RTW** pages of the **Simulation Parameters** dialog box. To access this dialog box, select **Parameters** from the Simulink **Simulation** menu:



Simulation time

Start time: 0.0 Stop time: 10.0

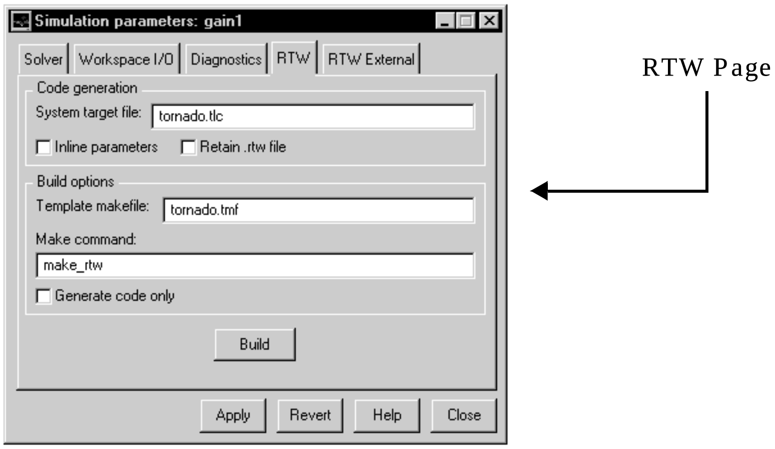
Solver options

Type: Fixed-step discrete (no continuous states)

Fixed step size: auto

Output options

Refine output Refine factor: 1



Code generation

System target file: tornado.tlc

☐ Inline parameters ☐ Retain .rtw file

Build options

Template makefile: tornado.tmf

Make command:

make_rtw

☐ Generate code only

For models with continuous blocks, set the **Type** to **Fixed-step**, the **Step Size** to the desired integration step size, and select the integration algorithm. For models that are purely discrete, set the integration algorithm to **discrete**.

Next, using the **RTW** page, set the **System target file** to `tornado.tlc`, set the **Template makefile** to specify the `tornado.tmf` template, which you have configured according to the instructions in “Configuring the Template Makefile” on page 8-12. (You can rename this file; simply change the dialog box accordingly.)

You can optionally inline parameters for the blocks in the C code, which can improve performance. Inlining parameters is not allowed when using external mode. See Chapter 4 for more information.

**Build Command Options.** You can specify four additional flags with the `make_rtw` command on the RTW page:

```
make_rtw EXT_MODE=1 STETHOSCOPE=1 MAT_FILE=1
         OPTS="-DMT -DSAVEFILE=file.mat -DVERBOSE"
```

These flags set the appropriate macros in the template makefile, causing any necessary additional steps to be performed during the build process.

- External mode — to enable the use of external mode in the generated executable, set `EXT_MODE=1`. You can optionally enable a verbose mode of external mode by appending `-DVERBOSE` to the `OPTS` flag. This causes parameter download information to be printed to the console of the VxWorks system.
- StethoScope — to enable the use of StethoScope with the generated executable, set `STETHOSCOPE=1`. There are two command line arguments, when starting `rt_main`, that control the block names used by StethoScope; you can use them when starting the program on VxWorks. See the next section, “Running the Program” on page 8-17 for more information on these arguments.
- MATLAB MAT-file — to enable data logging during program execution, set `MAT_FILE=1`. The program will create a file named `MODEL.mat` at the end of program execution; this file will contain the variables that you specified in the Solver page of the **Simulation Parameters** dialog box. By default, the MAT-file is created in the root directory of the current default device in VxWorks. This is typically the host file system that VxWorks was booted from. Other remote file systems can be used as a destination for the `.mat` file using rsh or ftp network devices or NFS. See the *VxWorks Programmer’s*

*Guide* for more information. If a device or filename other than the default is desired, add `"-DSAVEFILE=filename"` to the `OPTS` flag.

**Note:** When running in multitasking mode, MAT-file contents can vary from run to run. Logged variables in blocks with slower sample rates may change values at slightly different base sample times. This is due to the way multitasking works and does not imply incorrect results. To achieve identical results in the MAT-file from run to run or when comparing to Simulink, use singletasking mode.

- Multitasking mode — by default, the VxWorks model code is run in singletasking mode. Setting `OPTS=-DMT` causes the model to run in multitasking mode. Depending on the model, multitasking may provide the highest base sample rate, but may require additional blocks in the model in order to simulate correctly. See Chapter 7, “Models With Multiple Sample Rates,” for more information on singletasking vs. multitasking.

### Initiating the Build

Click on the **Build** button in the RTW page of the **Simulation parameters** dialog to build the program. The resulting object file is named with the `.lo` extension (which stands for “loadable object”). This file has been compiled for the target processor using the cross compiler specified in the makefile. If automatic downloading is enabled (`DOWNLOAD=yes`) in the template makefile, `tornado.tmf`, the target server is started and the object file is downloaded and started on the target. If StethoScope support was included in the build process, you can now start StethoScope on the host. The StethoScope object files, `libxdr.so`, `libutilstssip.so`, and `libscope.so`, must be loaded on the VxWorks target before the build is initiated or the automatic download will fail. See the *StethoScope User's Manual* for more information.

## Downloading and Running the Executable Interactively

If automatic downloading is disabled (DOWNLOAD=no), you must use the Tornado tools to complete the process. This involves three steps:

1. Establishing a communication link to transfer files between the host and the VxWorks target.
2. Transferring the object file from the host to the VxWorks target.
3. Running the program.

### Connecting to the VxWorks Target

After completing the build process, you are ready to connect the host workstation to the VxWorks target. The first step is starting the target server that is used for communication between the Tornado tools on the host and the target agent on the target. This is done either from the command line or from within the Tornado development environment. From the command line use:

```
tgtsvr target_network_name
```

### Downloading the Real-Time Program

To download the real-time program, use the VxWorks `ld` routine from within `WindSh`. `WindSh` (wind shell) can also be run from the command line or from within the Tornado development environment. (For example, if you want to download the file `vx_equal.lo`, which is in the `/home/my_working_dir` directory, use the following commands at the `WindSh` prompt:

```
cd "/home/my_working_dir"
ld <vx_equal.lo
```

You will also need to load the StethoScope libraries if the StethoScope option was selected during the build. The *Tornado User's Guide* describes the `ld` library routine.

### Running the Program

The real-time program defines a function, `rt_main()`, that spawns the tasks to execute the model code and communicate with Simulink (if you selected external mode during the build procedure.) It also initializes StethoScope if you selected this option during the build procedure.

The rt_main function is defined in the rt_main.c application module. This module is located in the *matlab*/rtw/c/tornado directory.

The rt_main function takes six arguments, and is defined by the following ANSI C function prototype:

```
SimStruct *rt_main(void (*model)(SimStruct *),
                   char *stop_time,
                   char *scopeInstallString,
                   int scopeFullNames,
                   int priority,
                   int TCPport);
```

The arguments are:

| | |
|---|---|
| model | A pointer to the entry point function in the generated code. This function has the same name as the Simulink model. It registers the local functions that implement the model code by adding function pointers to the model's SimStruct. See Chapter 6, "Program Architecture," for more information. |
| stop_time | The program stop time. This argument specifies the time to stop executing the model. Enter the stop time as a string since VxWorks does not allow arguments to be of type double in the taskSpawn function.<br><br>Specifying a stop time of "0.0" causes the model to execute indefinitely. Specifying a stop time of "-1.0" causes the model to use the stop time specified in the **Solver** page of the **Simulation Parameters** dialog box. |

| | |
|---|---|
| `scopeInstallString` | A character string that determines which signals are installed to StethoScope. Possible values are:<br>`NULL` (the default) — Install no signals.<br>`"*"` — Install all signals.<br>`"[A-Z]*"` — Install signals from blocks whose names start with a capital letter.<br>Specifying any other string installs signals from blocks whose names start with that string. |
| `scopeFullNames` | This argument determines whether StethoScope uses full hierarchical block names for the signals it accesses or just the individual block name. Possible values are:<br>`1` Use full block names<br>`0` Use individual block names (the default)<br>It is important to use full block names if your program has multiple instances of a model or S-function. |
| `priority` | The priority of the program's highest priority task (`tBaseRate`). Not specifying any value (or specifying a value of zero) causes `tBaseRate` to have the default priority of 30. |
| `TCPport` | The port number that the external mode sockets connection should use. The valid range is 256 to 65535 with 17725 as the default when nothing is specified. |

**Calling rt_main.** To begin program execution, call `rt_main` from `WindSh`. For example,

```
sp(rt_main, vx_equal, “0.0”, "*", 0, 30, 17725)
```

- Begins execution of the `vx_equal` model.
- Specifies no stop time so the program runs indefinitely.
- Provides access to all signals (block outputs) in the model by StethoScope.
- Uses only individual block names for signal access (instead of the potentially lengthy hierarchical name).
- Uses the default priority (30) for the `tBaseRate` task.
- Uses TCP port 17725, the default.

# Targeting DOS for Real-Time Applications

## Introduction

This chapter provides information that pertains specifically to using the Real-Time Workshop in a DOS environment.

This chapter includes a discussion of:

- DOS-based Real-Time Workshop applications
- Supported compilers and development tools
- Device driver blocks — adding them to your model and configuring them for use with your hardware
- Building the program

## Implementation Overview

The Real-Time Workshop includes DOS run-time interface modules designed to implement programs that execute in real-time under DOS. These modules, when linked with the code generated from a Simulink model, build a complete program that is capable of executing the model in real time. The DOS run-time interface files can be found in the *matlabroot*`/rtw/c/dos/rti` directory.

Real-Time Workshop DOS run-time interface modules and the generated code for the F-14 example from Chapter 2 are shown in the figure below.

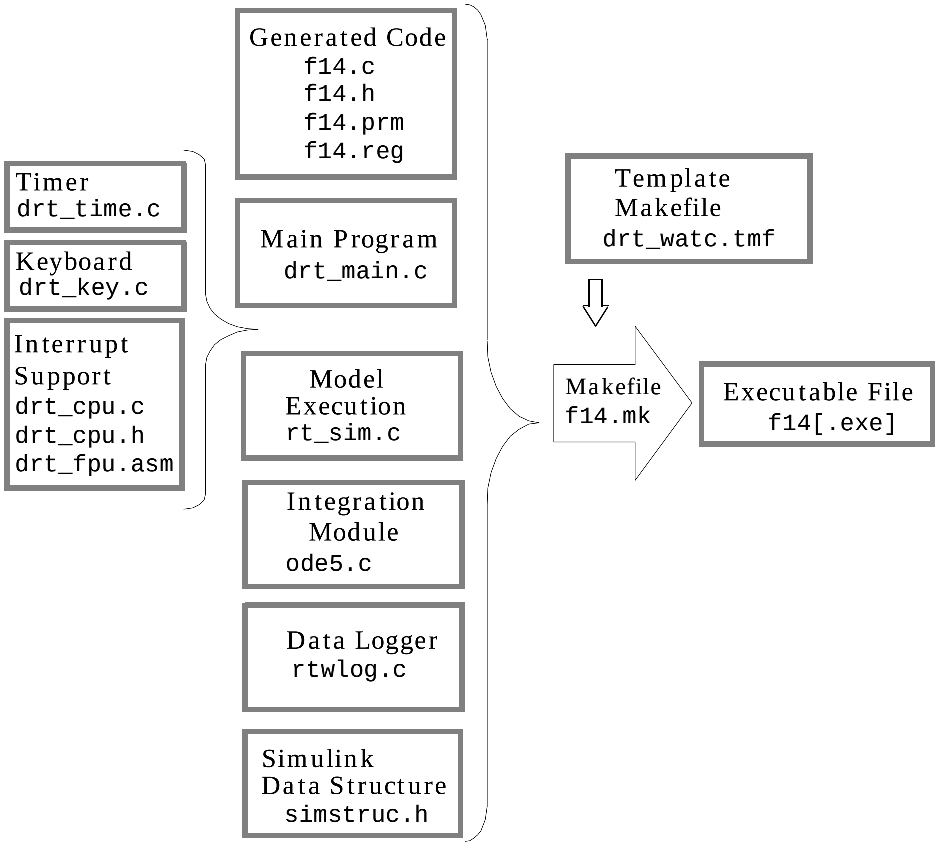


**Figure 9-1: Source Modules Used to Build the DOS Real-Time Program**

This diagram illustrates the code modules that are used to build a DOS real-time program.

To execute the code in real time, the program runs under the control of an interrupt driven timing mechanism. The program installs its own interrupt service routine (ISR) to execute the model code periodically at predefined sample intervals. The PC-AT's 8254 Programmable Interval Timer is used to time these intervals.

In addition to the modules shown in Figure 9-1, the DOS run-time interface also consists of device driver modules to read from and write to I/O devices installed on the DOS target.

Figure 9-2 shows the recommended hardware setup for designing control systems using Simulink, and then building them into DOS Realtime applications using RTW. The figure shows a robotic arm being controlled by a program (i.e., the controller) executing on the Target-PC. The controller senses the arm position and applies inputs to the motors accordingly, via the I/O devices on the target PC. The controller code executes on the PC and communicates with the apparatus it controls via I/O hardware.

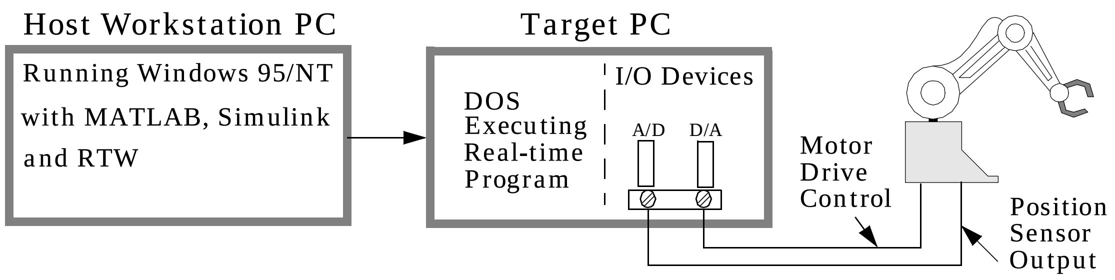


**Figure 9-2: Typical Hardware Setup**

## System Configuration

You can use the Real-Time Workshop with a variety of system configurations, as long as these systems meet the following hardware and software requirements.

### Hardware Requirements

The hardware needed to develop and run a real-time program includes:

- A workstation running Windows 95/NT and capable of running MATLAB/ Simulink. This workstation is the *host* where the real-time program is built.
- A PC-AT (386 or higher) running DOS. This system is the *target*, where the real-time program executes.
- I/O boards, which include analog to digital converter and digital to analog converters (collectively referred to as I/O devices), on the target.
- Electrical connections from the I/O devices to the apparatus you want to control (or to use as inputs and outputs to the program in the case of hardware-in-the-loop simulations).

Once built, you can run the executable on the target hardware as a stand-alone program that is independent of Simulink.

### Software Requirements

The development host must have the following software:

- MATLAB, Simulink to develop the model, and Real-Time Workshop to create the code for the model. You also need the run-time interface modules included with the Real-Time Workshop. These modules contain the code that handles timing, interrupts, data logging, and background tasks.
- Watcom C/C++ compiler, Version 10.6.

The target PC must have the following software:

- DOS4GW extender `dos4gw.exe`, included with your WATCOM Compiler package) must be on the search path on the DOS-targeted PC.

You can compile the generated code (i.e., the files *model*`.c`, *model*`.h`, etc.) along with user-written code using other compilers. However, the use of 16-bit compilers is not recommended for any application.

### make Utility

The Real-Time Workshop's automatic program builder for DOS targets uses the Watcom `make` utility `wmake`, to manage the process of building the program. `wmake` must be on the search path on the host workstation.

### Device Drivers

If your application needs to access its I/O devices on the target, then the real-time program must contain device driver code to handle communication with the I/O boards. RTW DOS run-time interface includes source code of the device drivers for the Keithley Metrabyte DAS 1600/1400 Series I/O boards. See the "Device Driver Blocks" section for information on how to use these blocks.

### Simulink Host

The development host must have Windows 95 or Windows NT to run Simulink. However, the real-time target requires only DOS, since the executable built from the generated code is not a Windows application. The real-time target will not run in a "DOS box" (i.e., a DOS window on Windows 95/NT).

Although it is possible to reboot the host PC under DOS for real-time execution, the computer would need to be rebooted under Windows 95/NT for any subsequent changes to the block diagram in Simulink. Since this process of repeated rebooting the computer is inconvenient, we recommend a second PC running only DOS as the real-time target.

## Sample Rate Limits

Program timing is controlled by installing an interrupt service routine that executes the model code. The target PC's CPU is then interrupted at the specified rate (this rate is determined from the step size).

The rate at which interrupts occur is controlled by application code supplied with the Real-Time Workshop. This code uses the PC-AT's 8254 Counter/Timer to determine when to generate interrupts.

The code that sets up the 8254 Timer is in `drt_time.c`, which is in the *matlabroot*`\rtw\c\dos\rti` directory. It is automatically linked in when you build the program using the DOS real-time template makefile.

The 8254 chip is a 16-bit counter that operates at a frequency of 1.193 MHz. However, the timing module, drt_time.c in the DOS run-time interface can extend the range by an additional 16 bits in software, effectively yielding a 32-bit counter. This means that the slowest base sample rate your model can have is

$$1.193\times10^{6} \div (2^{32} - 1) \approx \frac{1}{3600} Hz$$

This corresponds to a maximum base step size of approximately one hour.

The fastest sample rate you can define is determined by the minimum value from which the counter can count down. This value is 3, hence the fastest sample rate that the 8254 is capable of achieving is:

$$1.193 \times 10^6 \div 3 \approx 4 \times 10^5 \text{Hz}$$

This corresponds to a minimum base step size of

$$1 \div 4 \times 10^5 \approx 2.5 \times 10^{-6} \text{seconds}$$

However, bear in mind that the above number corresponds to the fastest rate the timer can generate interrupts. It does not account for execution time for the model code, which would substantially reduce the fastest sample rate possible for the model to execute in real time. Execution speed is machine dependent and varies with the type of processor and the clock rate of the processor on the target PC.

The slowest and fastest rates computed above refer to the base sample times in the model. In a model with more than one sample time, you can define blocks that execute at slower rates as long as the sample times are an integer multiple of the base sample time.

## Modifying Program Timing

If you have access to an alternate timer (e.g., some I/O boards include their own clock devices), you can replace the file `drt_time.c` with an equivalent file that makes use of the separate clock source. See the comments in `drt_time.c` to understand how the code works.

You can use your version of the timer module by redefining the `TIMER_OBJS` macros with the build command. For example, in the **RTW** page of the **Simulation parameters** dialog box, changing the build command to,

```
make_rtw TIMER_OBJS=my_timer.obj
```

replaces the file `drt_time.c` with `my_timer.c` in the list of source files used to build the program.

## Device Driver Blocks

The real-time program communicates with external hardware via a set of device drivers. These device drivers contain the necessary code for interfacing to specific I/O devices.

The Real-Time Workshop includes device drivers for commercially available Keithley Metrabyte 1600/1400 Series I/O boards. These device drivers are implemented as C-coded, S-functions to interface with Simulink. This means you can add them to your model like any other block.

In addition, each of these S-function device drivers has a corresponding target file to inline the device driver in the model code. See Chapter 10, “Targeting Custom Hardware,” for information on implementing your own device drivers.

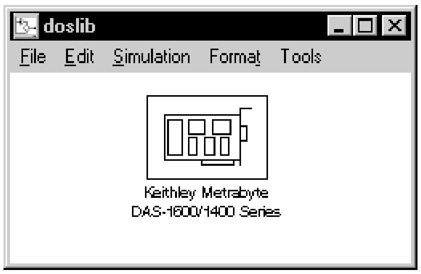
Since the device drivers are provided as source code, you can use these device drivers as a template to serve a a starting point for creating custom device drivers for other I/O boards.

### Device Driver Block Library

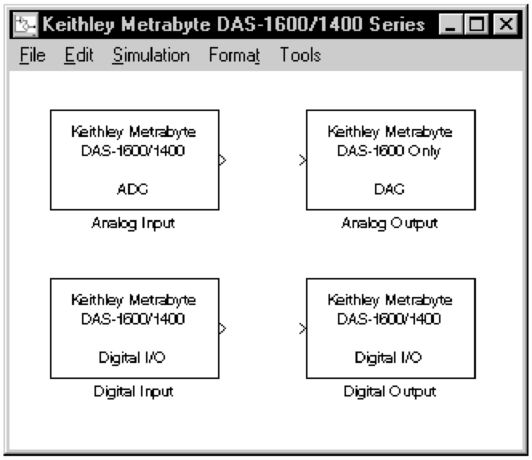
The device driver blocks for the Keithley Metrabyte 1600/1400 Series I/O boards designed for use with DOS applications are contained in a block library called `doslib` (*matlabroot*`\toolbox\rtw\doslib.mdl`). To display this library, type

```
doslib
```

at the MATLAB prompt. This window will appear:



To access the device driver blocks, double-click on the sublibrary icon.


Keithley Metrabyte DAS-1600/1400 Series
File Edit Simulation Format Tools
Keithley Metrabyte DAS-1600/1400 ADC
Analog Input
Keithley Metrabyte DAS-1600 Only DAC
Analog Output
Keithley Metrabyte DAS-1600/1400 Digital I/O
Digital Input
Keithley Metrabyte DAS-1600/1400 Digital I/O
Digital Output

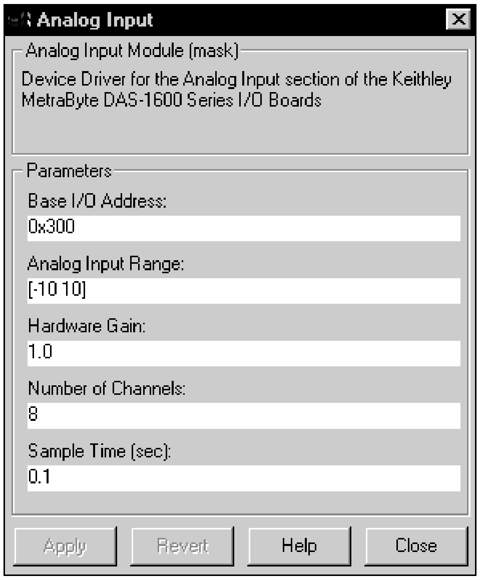
The blocks in the library contain device drivers that can be used for the DAS-1600/1400 Series I/O boards. The DAS-1601/1602 boards have 16 analog input (ADC) channels, two 12-bit analog output (DAC) channels and 4-bits of digital I/O. The DAS-1401/1402 boards do not have DAC channels. The DAS-1601/1401 boards have high programmable gains (1, 10, 100 and 500), while the DAS-1602/1402 boards offer low programmable gains (1, 2, 4 and 8). For more information, refer to the manufacturer's documentation for the I/O board.[1]

## Configuring Device Driver Blocks

Each device driver block has a dialog box that you use to set configuration parameters. As with all Simulink blocks, double-clicking on the block displays the dialog box. Some of the device driver block parameters (such as Base I/O Address) are hardware specific and are set either at the factory or configured via DIP switches at the time of installation.

1. *DAS-1600/1400 Series User's Guide, Revision B* - August, 1996. Part Number: 80940, Keithley Metrabyte Division, Keithley Instruments, Inc., 440 Myles Standish Blvd., Taunton, MA 02780. Website: `www.metrabyte.com`

## Analog Input (ADC) Block Parameters


Analog Input

Analog Input Module (mask)

Device Driver for the Analog Input section of the Keithley MetraByte DAS-1600 Series I/O Boards

Parameters

Base I/O Address:
0x300

Analog Input Range:
[-10 10]

Hardware Gain:
1.0

Number of Channels:
8

Sample Time (sec):
0.1

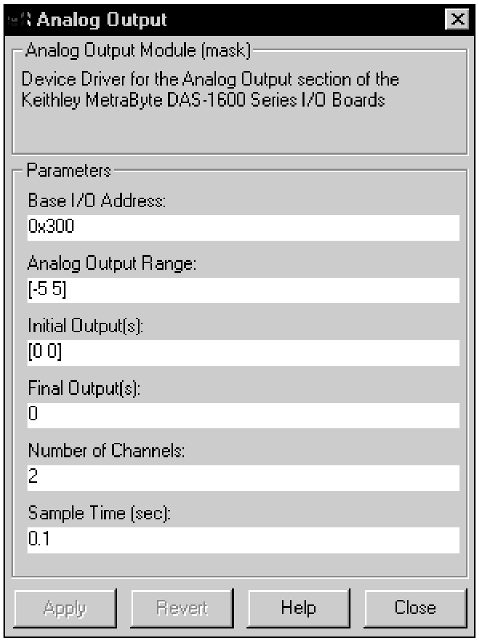
- **Base I/O Address** — The beginning of the I/O address space assigned to the board. The value specified here must match the board's configuration. Note that this parameter is a hexadecimal number and must be entered in the dialog as a MATLAB string (e.g., `'0x300'`).
- **Analog Input Range** — This two-element vector specifies the range of values supported by the Analog to Digital Converter. The specified range must match the I/O board's settings. Specifically, the DAS 1600/1400 Series boards can be switch configured to either `[0 10]` for unipolar or `[-10 10]` for bipolar input signals.
- **Hardware Gain** — This parameter specifies the programmable gain that is applied to the input signal before presenting it to the ADC. Specifically, the DAS-1601/1401 boards have programmable gains of 1, 10, 100, and 500. The DAS-1602/1402 boards have programmable gains of 1, 2, 4, and 8. Configure the Analog Input Range and the Hardware Gain depending on the type and range of the input signal being measured. For example, a DAS-1601 board in bipolar configuration with a programmable gain of 100 is best suited to measure input signals in the range between [±10v] ÷ 100 = ±0.1v.

  Voltage levels beyond this range will saturate the block output form the ADC block. Please adhere to manufacturers' electrical specifications to avoid damage to the board.
- **Number of Channels** — The number of analog input channels enabled on the I/O board. The DAS-1600/1400 Series boards offer up to 16 ADC channels when configured in unipolar mode (8 ADC channels if you select differential

mode). The output port width of the ADC block is equal to the number of channels enabled.

- **Sample Time (sec)** — Device drivers are discrete blocks that require you to specify a sample time. In the generated code, these blocks are executed at the specified rate. Specifically, when the ADC block is executed, it causes the ADC to perform a single conversion on the enabled channels, and the converted values are written to the block output vector.

## Analog Output (DAC) Block Parameters


Analog Output

Analog Output Module (mask)

Device Driver for the Analog Output section of the Keithley MetraByte DAS-1600 Series I/O Boards

Parameters

Base I/O Address: 0x300

Analog Output Range: [-5 5]

Initial Output(s): [0 0]

Final Output(s): 0

Number of Channels: 2

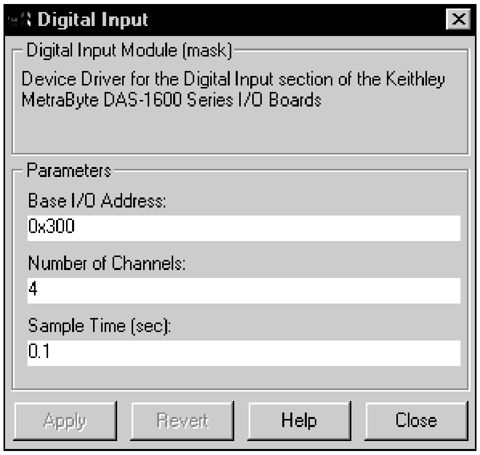
Sample Time (sec): 0.1

- **Base I/O Address** — The beginning of the I/O address space assigned to the board. The value specified here must match the board's configuration. Note that this parameter is a hexadecimal number and must be entered in the dialog as a MATLAB string (e.g., `'0x300'`).
- **Analog Output Range** — This parameter specifies the output range settings of the DAC section of the I/O board. Typically, unipolar ranges are between `[0 10]` volts and bipolar ranges are between `[-10 10]` volts. Refer to the DAS-1600 documentation for other supported output ranges.
- **Initial Output(s)** — This parameter can be specified either as a scalar or as an N element vector, where N is the number of channels. If a single scalar value is entered, the same scalar is applied to output. The specified initial output(s) is written to the DAC channels in the `mdlInitializeConditions` function.
- **Final Output(s)** — This parameter is specified in a manner similar to the Initial Output(s) parameter except that the specified final output values are

written out to the DAC channels in the `mdlTerminate` function. Once the generated code completes execution, the code sets the final output values prior to terminating execution.

- **Number of Channels** — Number of DAC channels enabled. The DAS-1600 Series I/O boards have two 12-bit DAC channels. The DAS-1400 Series I/O boards do not have any DAC channels. The input port width of this block is equal to the number of channels enabled.
- **Sample Time (sec)** — DAC device drivers are discrete blocks that require you to specify a sample time. In the generated code, these blocks are executed at the specified rate. Specifically, when the DAC block is executed, it causes the DAC to convert a single value on each of the enabled DAC channels, which produces a corresponding voltage on the DAC output pin(s).

### Digital Input Block Parameters


Digital Input

Digital Input Module (mask)

Device Driver for the Digital Input section of the Keithley MetraByte DAS-1600 Series I/O Boards

Parameters

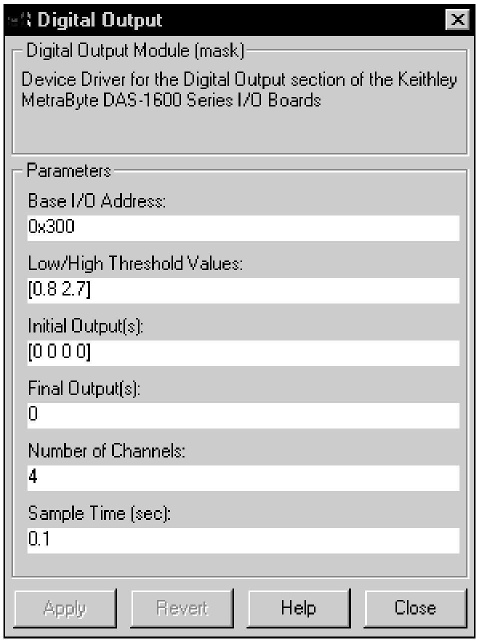
Base I/O Address: 0x300

Number of Channels: 4

Sample Time (sec): 0.1

- **Base I/O Address** — The beginning of the I/O address space assigned to the board. The value specified here must match the board's configuration. Note that this parameter is a hexadecimal number and must be entered in the dialog as a MATLAB string (e.g., `'0x300'`).
- **Number of Channels** — This parameter specifies the number of 1-bit digital input channels being enabled. This parameter also determines the output port width of the block in Simulink. Specifically, the DAS-1600/1400 Series boards provide four bits (i.e., channels) for digital I/O.
- **Sample Time (sec)** — Digital input device drivers are discrete blocks that require you to specify a sample time. In the generated code, these blocks are executed at the specified rate. Specifically, when the digital input block is executed, it reads a boolean value from the enabled digital input channels. The corresponding input values are written to the block output vector.

## Digital Output Block Parameters



Digital Output Module (mask)

Device Driver for the Digital Output section of the Keithley MetraByte DAS-1600 Series I/O Boards

Parameters

Base I/O Address:
0x300

Low/High Threshold Values:
[0.8 2.7]

Initial Output(s):
[0 0 0 0]

Final Output(s):
0

Number of Channels:
4

Sample Time (sec):
0.1

- **Base I/O Address** — The beginning of the I/O address space assigned to the board. The value specified here must match the board's configuration. Note that this parameter is a hexadecimal number and must be entered in the dialog as a MATLAB string (e.g., `'0x300'`).
- **Low/High Threshold Values** — This parameter specifies the threshold levels, `[lo hi]`, for converting the block inputs into 0/1 digital values. The signal in the block diagram connected to the block input should rise above the high threshold level for a 0 to 1 transition in the corresponding Digital Output Channel on the I/O board. Similarly, the input should fall below the low threshold level for a 1 to 0 transition.
- **Initial Output(s)** — Same as the Analog Output block, except the specified values are converted to 0 or 1 based on the lower threshold value before they are written to the corresponding digital output channel.
- **Final Output(s)** — Same as the Analog Output block, except the specified values are converted to 0 or 1 based on the lower threshold value before they are written to the corresponding digital output channel on the I/O board.
- **Number of Channels** — This parameter specifies the number of 1-bit digital I/O channels being enabled. This parameter also determines the output port width of the block. Specifically, the DAS-1600/1400 Series boards provide four bits (i.e., channels) for digital I/O.
- **Sample Time (sec)** — Digital output device drivers are discrete blocks that require you to specify a sample time. In the generated code, these blocks are executed at the specified rate. Specifically, when the digital output block is

executed, it causes corresponding boolean values to be output from the board's digital I/O channels.

## Adding Device Driver Blocks to the Model

Add device driver blocks to the Simulink block diagram as you would any other block — simply drag the block from the block library and insert it into the model. Connect the ADC or Digital Input module to the model's inputs and connect the DAC or Digital Output module to the model's outputs.

### Including Device Driver Code

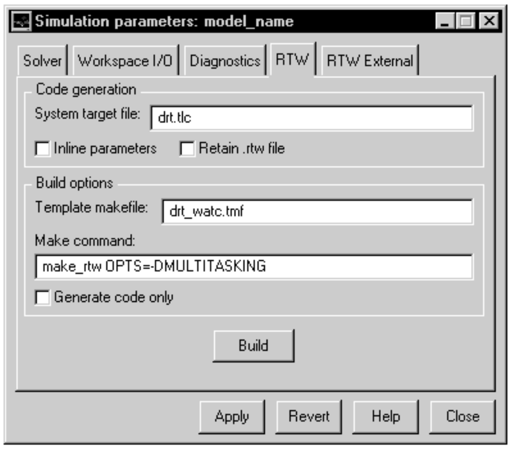
Device driver blocks are implemented as S-functions written in C. The C code for a device driver block is compiled as a MEX-file so that it can be called by Simulink. See the *Application Program Interface Guide* for information on MEX-files.

The same C code can also be compiled and linked to the generated code just like any other C-coded, S-function. However, by using the target (`.tlc`) file that corresponds to each of the C file S-functions, the device driver code is inlined in the generated code.

The *matlabroot*`\rtw\c\dos\devices` directory contains the MEX-files, C files, and target files (`.tlc`) for the device driver blocks included in `doslib`. This directory is automatically added to your MATLAB path when you include any of the blocks from `doslib` in your model.

## Building the Program

Once you have created your Simulink model and added the appropriate device driver blocks, you are ready to build a DOS target application. To do this, select **Parameters** from the **Simulation** menu of your Simulink model and display the RTW page of the **Simulink parameters** dialog by clicking on its tab.



System target file: drt.tlc
☐ Inline parameters ☐ Retain .rtw file
Template makefile: drt_watc.tmf
Make command: make_rtw OPTS=-DMULTITASKING
☐ Generate code only

On the RTW page, specify:

- `drt.tlc` as the system target file
- `drt_watc.tmf` as the template makefile. This is used with the Watcom C/386 compiler, assembler, linker, and `WMAKE` utility.

You can specify TLC options in the System target file field following `drt.tlc`, and make options in the Make command field following `make_rtw`. Chapters 2 and 3 provide detailed descriptions of the available TLC and make options. The DOS system target file, `drt.tlc`, and the template makefile, `drt_watc.tmf`, are located in the `matlab\rtw\c\dos` directory.

1 The template makefile assumes that the Watcom C/386 Compiler, Assembler, and Linker have been correctly installed on the host workstation. You can verify this by checking the environment variable,

`WATCOM`, which correctly points to the directory where the Watcom files are installed.

2 The program builder invokes the Watcom `wmake` utility on the generated makefile, so the directory where `wmake` is installed must be on your path.

## Running the Program

The result of the build process is a DOS 32-bit protected-mode executable. The default name of this is *`model`*`.exe`, where *`model`* is the name of your Simulink model. You must run this executable in DOS; you cannot run the executable in Windows 95/NT.

# Targeting Custom Hardware

## Introduction

This chapter contains information on targeting custom hardware and implementing device driver blocks. By implementing your own blocks, you can create a `boardlib` library to include blocks for your particular I/O devices.

To target custom hardware, you must:

- Create a run-time interface for your target system to execute the generated code.
- Create a system target file. This the entry point for the TLC program used to transform the models into generated code. The system target file can then use the block target files and TLC function library provided by The MathWorks. Or you can create your own block target files and TLC function library.
- Create a template makefile to build your real-time executable.
- Write device drivers. Device drivers consist of a Simulink C MEX S-function and, optionally, a corresponding target file to inline the device instructions.

This chapter provides information on:

- Targeting custom hardware
- What a device driver block is
- The structure of an S-function
- Functions required by the S-function API
- How to implement the operations performed by a device driver
- Compiling a device driver as a MEX-file
- Masking a device driver block

Device driver blocks can be implemented as S-functions. This chapter assumes that you are familiar with the Simulink C-MEX S-function format and API. See the Simulink documentation for more information on S-functions.

You can implement device driver blocks in two ways:

1. As C language S-functions (with no TLC files provided). This is referred to as a *noninlined* S-function.

2. As inlined S-functions (using TLC files).

## Run-Time Interface

To create a run-time interface, you can begin with the generic real-time target (`grt`). The run-time interface for `grt` consists of

- `grt_main.c` — located in *matlabroot*`/rtw/c/grt`.
- `rt_sim.c` — located in *matlabroot*`/rtw/c/src`.
- `ode1.c` - `ode5.c` — you must use one of these solvers if your model has continuous states. These are located in *matlabroot*`/rtw/c/src`.
- library routines for the model code — located in *matlabroot*`/rtw/c/libsrc`.

For your run-time interface, you will need to copy and modify `grt_main.c`. The other modules do not require modification.

To create a new target called "mytarget", you start by creating this directory:

*matlabroot*`/rtw/c/mytarget`

Copy `grt_main.c` to this directory and rename it `rt_main.c`. You will then need to modify `rt_main.c` to execute your model code. By default, `grt_main.c` is set up to execute your code in pseudomultitasking mode. If your target has a real-time operating system, you will need to modify rt_main.c to include calls for task creation and management. See Chapter 5, "Model Code," Chapter 6, "Program Architecture," and Chapter 7, "Models With Multiple Sample Rates," for more details on the model code and how it is executed. The Tornado target is an example of a system that has a real-time operating system (see *matlabroot*`/rtw/c/tornado/rt_main.c` for the Tornado main module). Refer to Chapter 8, "Targeting Tornado for Real-Time Applications," for detailed information about targeting Tornado.

## Creating System Target Files and Template Makefiles

Assuming that you've created the directory

```
matlabroot/rtw/c/mytarget
```

you should copy *matlabroot*/rtw/c/grt/grt.tlc into it and rename grt.tlc to mytarget.tlc. You can then modify your system target file as needed, based on the requirements of your real-time target.

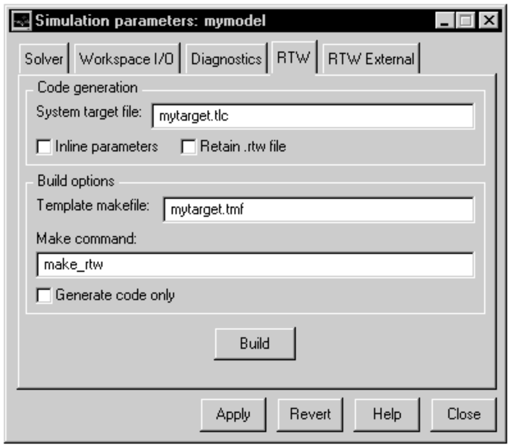
Chapter 3, "Code Generation and the Build Process," provides information on how makefiles work. If your compiler and linker come with a make utility, then you can use it providing that it conforms to the "general" structure of most make utilities. If a make utility isn't provided with your compiler, you can use GNU make, which is located in rtw/bin/arch/make. When working with GNU make, you should use grt_unix.tmf as a starting point — even on PC platforms.

Once you've created the directory

```
matlabroot/rtw/c/grt
```

you should copy one of the *matlabroot*/rtw/c/grt/*.tmf template makefiles to this directory and rename it mytarget.tmf. You will need to modify the template makefile to support your compiler, linker, and your version of make.

You can exercise mytarget.tlc and mytarget.tmf by creating a small Simulink fixed-step model. In the RTW page of the Simulation parameters dialog box, you must set the System Target file to *mytarget*`.tlc` and the template makefile to *mytarget*`.tmf`:



Code generation

System target file: mytarget.tlc

☐ Inline parameters ☐ Retain .rtw file

Build options

Template makefile: mytarget.tmf

Make command: make_rtw

☐ Generate code only

Clicking the **Build** button causes the Real-Time Workshop to generate code as dictated by *mytarget*`.tlc` and compile it as dictated by *mytarget*`.tmf`.

## Implementing Device Drivers

S-functions can be built into MEX-files or compiled and linked with other code to become part of a stand-alone program. This dual nature is exploited by the device driver blocks. These blocks are implemented as C code S-functions. They are compiled as MEX-files so you can place them in your Simulink block diagram. When you use the same model to build a real-time program, the source code for the device driver blocks is automatically compiled and linked with your program.

Additionally, the Real-Time Workshop uses Target Language Compiler technology that allows you to *inline* your S-function. By providing a customized TLC file containing the definition of your block, you can directly generate the code for your device driver. The advantage of inlining your S-function in this fashion is the elimination of function call overhead associated with the S-function interface.

It is helpful to examine existing device driver code in conjunction with this chapter. The source code for the blocks in block library resides in the following directories:

- *matlabroot*`/rtw/c/dos/devices` directory for the source code to blocks in `doslib`.
- *matlabroot*`/rtw/c/tornado/devices` directory for the source code to blocks in `vxlib`.

There is also an S-function template that provides a useful starting point for the creation of any S-function. The file is:

*matlabroot*`/simulink/src/sfuntmpl.c`

or

*matlabroot*`/simulink/src/sfuntml.doc`

---

**Note**: This chapter does not discuss inlining of S-functions. Refer to “Inlining an S-function” in the *Target Language Compiler Reference Guide.*

---

Device drivers are implemented as C MEX S-functions. When you insert these blocks into your Simulink model, the code generator compiles and links the block's source along with the source files used to build your program.

You can add your own blocks to a device driver library by writing S-functions that implement device drivers for your particular I/O board. However, in order for the code generator to generate code that can call your blocks, they must be implemented using the API defined by Simulink.

Before beginning the task of creating your own device driver block, you should consult the Simulink documentation for a description of how to write S-functions using the API.

## Device Driver Blocks

A device driver block is:

- A device driver — software that handles communication between a real-time program and an I/O device. See your particular hardware documentation for information on its requirements.
- An S-function — a system function that interacts with Simulink during the simulation and is linked with the generated code. See the Simulink documentation for more information.
- A C MEX-file — a C subroutine that is dynamically linked to MATLAB. See the *MATLAB Application Program Interface Guide* for more information on MEX-files.
- A masked Simulink block — masked blocks allow you to define your own dialog box, icon, and initialization commands for the block. See the *Using Simulink* manual for more information on masking blocks.

## The S-Function Format

The S-function API requires you to define specific functions and a `SimStruct` (a Simulink data structure). Like the generated code, the functions that implement the S-function are private to the source file. This means you can generate code for models having multiple instances of the same S-function.

Device driver S-functions are relatively simple to implement because they perform only a few operations. These operations include:

- Initializing the `SimStruct`
- Initializing the I/O device
- Calculating the block outputs according to the type of driver being implemented:

  Reading values from an I/O device and assigning these values to the block's output vector `y` (if it is an ADC)

  Writing values from the block's input vector `u`, to an I/O device (if it is a DAC)
- Terminating the program (e.g., zeroing the DAC outputs)

Regardless of their simplicity, all S-functions must contain the full set of functions required by the S-function API. Unused functions are implemented as empty stubs.

### Required Functions

The S-function API requires you to define these functions:

- Functions called during initialization:

  `sfunction_name` (the public registration function)
  `mdlInitializeSizes`
  `mdlInitializeSampleTimes`
  `mdlInitializeConditions`

- Function called to calculate block outputs:

  `mdlOutputs`

- Functions not used, but included as stubs:

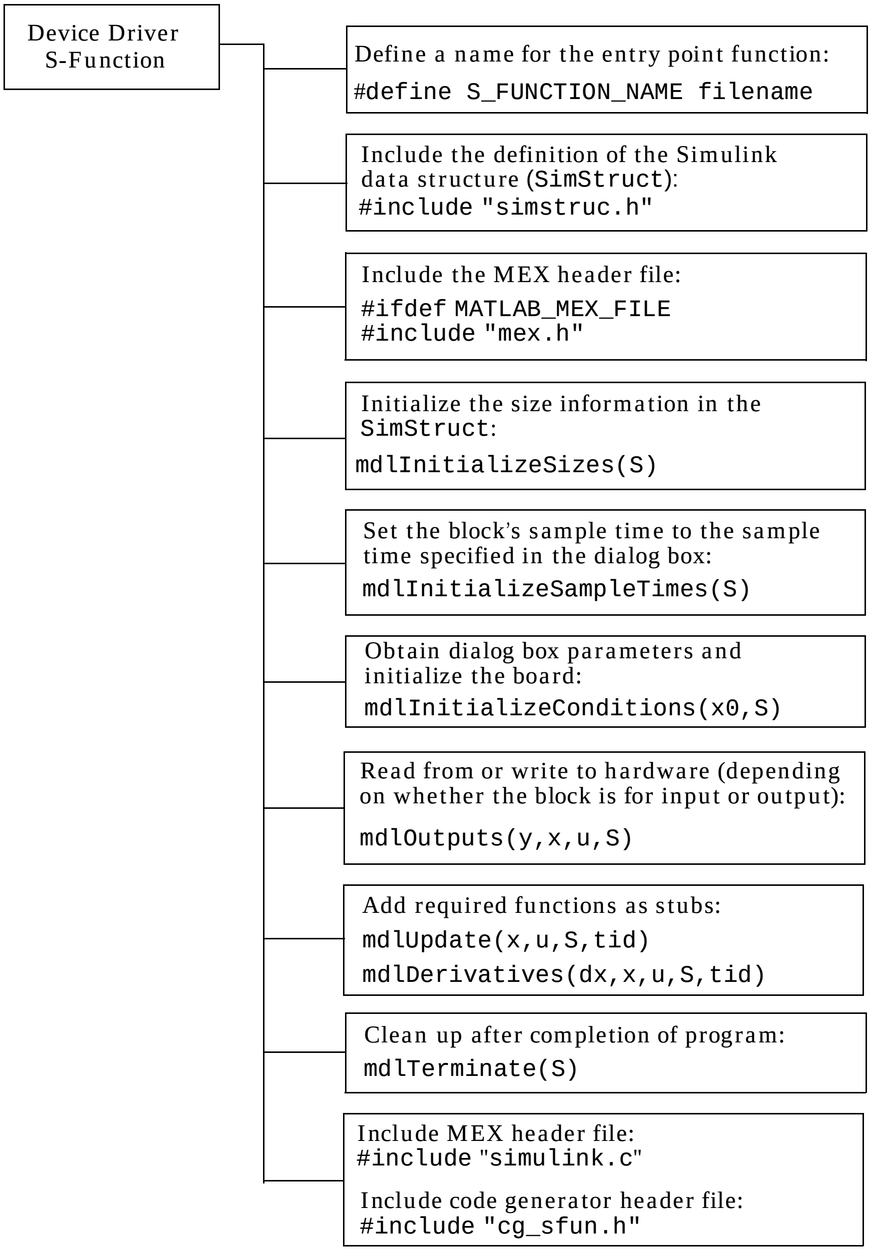
  `mdlUpdate`
  `mdlDerivatives`
- Function called to reset the hardware when the program terminates:

  `mdlTerminate`

The sections that follow describe how to implement these functions.

## S-Function File Format

S-functions also require certain defined values and include files. The diagram on the next page illustrates the overall format of a device driver S-function.



Device Driver S-Function

Define a name for the entry point function:

```
#define S_FUNCTION_NAME filename
```

Include the definition of the Simulink data structure (`SimStruct`):

```
#include "simstruc.h"
```

Include the MEX header file:

```
#ifdef MATLAB_MEX_FILE
#include "mex.h"
```

Initialize the size information in the `SimStruct`:

```
mdlInitializeSizes(S)
```

Set the block's sample time to the sample time specified in the dialog box:

```
mdlInitializeSampleTimes(S)
```

Obtain dialog box parameters and initialize the board:

```
mdlInitializeConditions(x0,S)
```

Read from or write to hardware (depending on whether the block is for input or output):

```
mdlOutputs(y,x,u,S)
```

Add required functions as stubs:

```
mdlUpdate(x,u,S,tid)
mdlDerivatives(dx,x,u,S,tid)
```

Clean up after completion of program:

```
mdlTerminate(S)
```

Include MEX header file:

```
#include "simulink.c"
```

Include code generator header file:

```
#include "cg_sfun.h"
```

**Figure 10-1: Format of a Device Driver S-Function**

### The S_FUNCTION_NAME Definition

The statement

```
#define S_FUNCTION_NAME name
```

defines the name of the function that is the entry point for the S-function code. This function is defined in `cg_sfun.h`, which is included at the end of the S-function. Note that `name` must be the name of the S-function file without the `.c` extension (e.g., `dt2811ad` for the Data Translation DT2811 A/D block). Also, you must specify the `S_FUNCTION_NAME` before you include `simstruc.h`.

### Defining the SimStruct

You must include the file `simstruc.h` to define the `SimStruct` (the Simulink data structure) and the `SimStruct` access macros:

```
#include "simstruc.h"
```

`simstruc.h` itself includes `rt_matrx.h`, which contains definitions for `mxGetPr` (and other matrix access macros). These `mx` macros are equivalent to those with the same name defined in MATLAB's Application Program Interface Library. They are used to obtain the parameters specified in the device driver's dialog box. See the *Application Program Interface Guide* for more information on matrix access macros.

This parallel definition of matrix access macros allows the same source code to be used as a MEX-file or as a source module that is compiled and linked with the generated code.

## Conditional Compilations

In general, you can use an S-function in these environments:

- Simulink
- Real-Time

When you use your S-function in Simulink, you do so by creating a MEX-file from it. In this case, the macro `MATLAB_MEX_FILE` is defined.

When you use your S-function in real-time (for example, with a fixed-step solver and the Real-Time Workshop), the macro `RT` is defined.

## Initialization

Initialization is performed in three separate steps. The S-function API requires you to implement three functions for initialization:

- `mdlInitializeSizes` — specifies the sizes of various parameters in the `SimStruct`.
- `mdlInitializeSampleTimes` — obtains the board's sample time from the dialog box and sets this value in the `SimStruct`.
- `mdlInitializeConditions` — reads values from the dialog box, initializes the board, and saves parameter values in the `SimStruct`.

### Obtaining Values from Dialog Boxes

Obtaining the values of parameters from the block's dialog box requires you to:

1. Get the S-function input parameter (in this case the dialog box entry) using the `ssGetSFcnParam` macro.

2. Get the particular value from the input `mxArray` (since all input arguments to MEX-files are of type `Mxarray *`) using the `mxGetPr` function.

For example, suppose you want the value of the third item in the block's dialog box, which happens to be an integer. For convenience, you can define a macro that obtains the desired argument:

```
#define THIRD_ARGUMENT ssGetSFcnParam(S, 2)
```

Next define a variable used to store the integer (which is the number of channels in this example):

```
int num_channels;
```

Finally, extract the value from the argument (which is of type `mxArray *`), and assign it to a type `int`. Since the parameter is a single integer, you need only the first element in the parameter:

```
num_channels = mxGetPr(THIRD_ARGUMENT)[0];
```

The macro `ssGetSFcnParam` is part of the S-function API. It is defined in `simstruc.h` and described in the *Using Simulink* manual.

The function mxGetPr is part of the External Interface Library. Its function prototype is in mex.h and rt_matrx.h. It is described in the *Application Program Interface Guide*.

### Initializing Sizes — Input Devices

The mdlInitializeSizes function sets size information in the SimStruct. For example, the following definition of mdlInitializeSizes initializes a typical ADC (analog to digital converter) board:

```
static void mdlInitializeSizes(SimStruct *S)
{
  ssSetNumSFcnParams(S, 0);  /* Number of expected parameters */
  if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
/* Return if number of expected != number of actual parameters */
    return;
  }
  ssSetNumContStates(S, 0);
  ssSetNumDiscStates(S, 0);
  ssSetNumInputs(S, 0);
  ssSetNumOutputs(S, mxGetPr(ssGetArg(S,0)[0]);
  ssSetDirectFeedThrough(S, 0);
  ssSetNumInputArgs(S, NUMBER_OF_ARGS);
  ssSetNumSampleTimes(S, 1);
  ssSetNumIWork(S, NUMBER_OF_IWORKS);
  ssSetNumRWork(S, NUMBER_OF_RWORKS);
}
```

This list describes the function of each macro in mdlInitializeSizes:

- ssSetNumSFcnParams — The number of input parameters is equal to the number of parameters in the block's dialog box.
- ssSetNumContStates, ssSetNumDiscStates — There are no continuous or discrete states, so these sizes are set to 0.
- ssSetNumInputs — The ADC block has no inputs because it reads data from the I/O board. An ADC is a *source* block (i.e., the block has only output ports).
- ssSetNumOutputs — The number of outputs equals the number of I/O channels. The code above obtains the number of channels from the dialog

box, which is the first argument passed to the S-function (i.e., the first item in the dialog box).

- ssSetDirectFeedThrough — It is important to note that the ADC has no direct feedthrough. The ADC's output is calculated based on values obtained external to the Simulink model, not from the input of another block. In fact, the ADC has no inputs; it calculates its output based on values obtained from the I/O board. This information affects the organization of the generated code.
- ssSetNumSampleTimes — There is only one sample time (the board's sample time specified in its dialog box).
- ssSetNumIWork — One IWork (temporary storage in the SimStruct) is allocated for each integer parameter in the dialog box.
- ssSetNumRWork — One RWork (temporary storage in the SimStruct) is allocated for each real parameter in the dialog box.

### Temporary Storage for Parameters

There are no specific fields in the SimStruct designated for the storage of most device driver parameters (sample time is an exception; see the "Initializing Sample Times" section). However, the SimStruct does contain temporary storage for integers, reals, and pointers, which is referred to as the IWork, RWork, PWork vectors, respectively.

It is necessary to use this temporary storage since each function within the S-functions uses local variables. Using local *static* storage would make your S-function nonreentrant, which means you could not have multiple instances of it in the same model.

For example, to save the num_channels variable obtained from the dialog box (see the "Obtaining Values from Dialog Boxes" note) in the SimStruct, use the S-function API macro ssSetIWorkValue:

```
#define NUM_CHANNELS 0
ssSetIWorkValue(S, NUM_CHANNELS, num_channels);
```

This saves the number of channels in the first element of the IWork vector. To retrieve the value from within another function, use the ssGetIWorkValue macro:

```
num_channels = ssGetIWorkValue(S, NUM_CHANNELS);
```

Note that `NUM_CHANNELS` is the index into the `IWork` vector.

### Initializing Sizes — Output Devices

Initializing size information for an output device, such as a DAC (digital to analog converter) has two important differences from the ADC:

- Since the DAC obtains its inputs from other blocks, the number of channels is equal to the number of inputs (instead of number of outputs as is the case with the ADC). This is because the DAC is a *sink* block (i.e., the block has only input ports and it writes data to something that is external to the Simulink model — the I/O device).
- Also notice that this block has direct feedthrough (i.e., the DAC cannot execute until the block feeding it updates its outputs). This information affects the organization of the generated code.

The following example illustrates the definition of `mdlInitializeSizes` for a DAC:

```
static void mdlInitializeSizes(SimStruct *S)
{
  ssSetNumSFcnParams(S, 0);  /* Number of expected parameters */
  if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
/* Return if number of expected != number of actual parameters */
    return;
  }
  ssSetNumContStates(S, 0);
  ssSetNumDiscStates(S, 0);
  ssSetNumInputs(S, mxGetPr(ssGetArg(S,0)[0]);  /*Number of
                       inputs is now the number of channels. */
  ssSetNumOutputs(S, 0);
  ssSetDirectFeedThrough(S, 1);  /* This block has direct
                                              feedthrough.*/
  ssSetNumSampleTimes(S, NSAMPLE_TIMES);
  ssSetNumIWork(S, NUMBER_OF_IWORKS);
  ssSetNumRWork(S, NUMBER_OF_RWORKS);
}
```

### Initializing Sample Times

Device driver blocks are discrete blocks that require you to set a sample time. The S-function has only one sample time — the sample time specified in the dialog box.

The following definition of `mdlInitializeSampleTimes` reads the sample time from the block's dialog box. In this case, sample time is the second parameter in the dialog box:

```
static void mdlInitializeSampleTimes(SimStruct *S)
{
  ssSetSampleTimeEvent(S, 0, mxGetPr(ssGetSFcnParams(S,1))[0]);
  ssSetOffsetTimeEvent(S, 0, 0.0);
}
```

### Initializing the I/O Device

Device driver S-functions use the `mdlInitializeConditions` function to:

- Read parameters from the block's dialog box
- Save parameter values in the `SimStruct`
- Initialize the I/O device

When reading parameter values, you must be careful to handle data types correctly. The next two examples illustrate how to read two different types of data from the dialog box.

**Reading the Base Address.** The following code reads the base address parameter from the dialog box. While this parameter is actually a hexadecimal number, it is specified as a character string and is ultimately stored as an integer:

```
static void mdlInitializeConditions(real_T *x0,
                                    SimStruct *S)
{
  int base_addr_str_len = 128;
  char base_addr_str[128];
  mxGetString(ssGetSFcnParams(S, 0), base_addr_str,
  base_addr_str_len);
  sscanf(base_addr_str, "%lx", &base_addr);
  ssSetIWorkValue(S, 0, (int) base_addr);
}
```

In this example, the base address is the first argument in the dialog box. You obtain the parameter with ssGetArg and then use mxGetString to convert the argument (which, like all MEX-file arguments, is actually a type mxArray *) to a string. The parameter is passed as a string because you cannot specify hexadecimal numbers in the dialog box.

The conversion from string to hexadecimal is accomplished by sscanf, which returns the integer equivalent of the hexadecimal number. ssSetIWorkValue then stores this integer as the first element in the IWork vector in the SimStruct.

**Reading the Number of Channels.** Add the following code to mdlInitializeConditions to read the number of channels and save this integer value in the SimStruct. In this case, the number of channels is not allowed to exceed 4. The value obtained is set to the second element in the IWork vector:

```
int_T i;
int_T num_channels = min(4,
  (int) mxGetPr(ssGetSFcnParams(S,0))[0]);
ssSetIWorkValue(S, 1, num_channels);
```

**Zeroing the Board Output.** You may also want to zero the output of the DAC to prevent erratic output before the first conversion. For example, from the dSPACE DS1102 DAC:

```
#ifdef RT
/* Zero the DACs */
  for (i = 0; i < num_channels; i++) {
    ds1102_da(i + 1, 0.0);
  }
#endif
}
```

This code zeros the output by calling the hardware-specific function ds1102_da for each channel and writing the value 0.0 to the board. Note that these statements are enclosed in the “if defined MATLAB_MEX_FILE” conditional statements. This particular set of drivers is not intended to be used during simulation, but only by the real-time program.

For an example of device drivers designed to be used for real-time programs as well as during a Simulink simulation, see the doslib block library.

## Calculating Block Outputs

The basic purpose of a device driver block is to allow your program to communicate with I/O hardware. Typically, you can accomplish this using low level hardware calls that are part of your compiler's C library or using C-callable functions provided with your I/O hardware. This section provides examples of both techniques.

All S-functions call the `mdlOutputs` function to calculate block outputs. For a device driver block, this function contains the code that reads from or writes to the hardware.

### Accessing the Hardware

The mechanism you use to access the I/O device depends on your particular implementation. One possibility is to use low level hardware calls that are part of your compiler's C library. For example, on the PC a device driver could use the following technique. These S-functions define access routines for both of the supported compilers:

```
/* compiler dependent low level hardware calls */
#ifdef __HIGHC__
#include <conio.h>
#define hw_outportb(portid, value) _outp(portid, value)
#define hw_inportb(portid)         _inp(portid)
#define hw_outport(portid, value)  _outpw(portid, value)
#define hw_inport(portid )         _inpw(portid)
#elif __WATCOMC__
#include <conio.h>
#define hw_outportb(portid, value) outp(portid, value)
#define hw_inportb(portid)         inp(portid)
#define hw_outport(portid, value)  outpw(portid, value)
#define hw_inport(portid)          inpw(portid)
#else
```

Another technique is to use C callable functions provided with the I/O hardware.

### ADC Outputs

In the case of an ADC, `mdlOutputs` must:

- Initiate a conversion for each channel.
- Read the board's A/D converter output for each channel (and perhaps apply scaling to the values read).
- Set these values in the output vector `y` for use by the model.

For example, the following code is from the dSPACE DS1102 ADC:

```
static void mdlOutputs(real_T *y, real_T *x, real_T *u,
                       Simstruct *S, int tid)
{
  int num_channels = ssGetIWorkValue(S, 0);
  int i;
  ds1102_ad_start(); /* Start the four ADC's */
  for (i = 0; i < num_channels; i++) {
    y[i] = ds1102_ad(i + 1) * ADC_GAIN;
  }
}
```

### DAC Outputs

In the case of the DAC, the block inputs (i.e., the model outputs) are read from the block input vector `u` and written to the I/O device. Using the DS1102 again as an example, replace the `for` loop in the ADC code with:

```
for (i = 0; i < num_channels; i++) {
  value = u[i];
    if (value > DAC_MAX_VOLTAGE) {
      value = DAC_MAX_VOLTAGE;
    }
  if (value < DAC_MIN_VOLTAGE) {
    value = DAC_MIN_VOLTAGE;
  }
  ds1102_da(i + 1, value * DAC_GAIN);
}
```

The values written to the DAC are kept within an acceptable range by comparing them to predefined minimum and maximum values.

The two preceding examples use hardware specific functions to access the boards. The `ds1102_ad_start` function initiates the analog to digital conversion for each channel. The `ds1102_ad` reads from the ADC and the `ds1102_da` writes to the DAC.

## Unused but Required Functions

To adhere to the Simulink S-function format, you must include two additional functions that are not used, but must be defined as empty stubs:

```
static void mdlUpdate(real_T *x, real_T *u, SimStruct *S,
                      int_T tid)
{
}
static void mdlDerivatives(real_T *dx, real_T *x, realT *u,
                           SimStruct *S, int_T tid)
{
}
```

Note that these stubs are included in the S-function template file *matlabroot*/`simulink/src/sfuntmpl.c`, which provides an ideal starting point for implementing your own device drivers.

## The Termination Function

The final required function is typically used only in DACs to zero the output at the end of the program. For example:

```
static void mdlTerminate(SimStruct *S)
{
int num_channels = ssGetIWorkValue(S, 0);
int i;
for (i = 0; i < num_channels; i++) {
  ds1102_da(i + 1, 0.0);
  }
}
```

This `for` loop simply sets the output of each channel to 0.0. ADCs typically implement this function as an empty stub.

## Additional Include Files

The code implementing a device driver block must serve as both MEX-file and as stand-alone code module that can be compiled and linked with the generated code. This code is used as a MEX-file to enable Simulink to obtain information for performing consistency checks during the code generation process.

Each of these applications of the device driver S-function requires its own include file. The two files are `simulink.c` for MEX-files and `cg_sfun.h` for the generated code.

MEX-files are built with the `mex` command. `mex` defines the string `MATLAB_MEX_FILE` to provide a mechanism that allows you to include certain parts of your code in the MEX-file and other parts in the generated code. To include the proper files, place the following statements at the end of your S-function:

```
#ifdef MATLAB_MEX_FILE /* Is this a MEX-file? */
#include "simulink.c" /* MEX-file include file */
#else
#include "cg_sfun.h"  /* Generated code include file */
#endif
```

The S-function template file *matlabroot*`/simulink/src/sfuntmpl.c` contains these statements.

### The Public Registration Function

The include file `cg_sfun.h` defines a function that is the entry point for the S-function code. This function is named by the

```
#define S_FUNCTION_NAME name
```

macro that you specified earlier in your code.

This function registers the other local functions by installing pointers to them in the `SimStruct`. This is the only public function in the S-function and is called by the real-time program's public registration function during program startup. The remainder of the S-function code is then accessed via these pointers. This scheme produces re-entrant code allowing multiple instances of an S-function within a single program.

The S_FUNCTION_NAME function always has the same definition and therefore is most simply implemented by including cg_sfun.h:

```
void S_FUNCTION_NAME(SimStruct *S){
ssSetmdlInitializeSizes(S, mdlInitializeSizes);
ssSetmdlInitializeConditions(S, mdlInitializeConditions);
ssSetmdlInitializeSampleTimes(S,mdlInitializeSampleTimes);
ssSetmdlOutputs(S, mdlOutputs);
ssSetmdlUpdate(S, mdlUpdate);
ssSetmdlDerivatives(S, mdlDerivatives);
ssSetmdlTerminate(S, mdlTerminate);
}
```

Note that this function is directly analogous to the MODEL_NAME function in the generated code. See “model.reg” on page 5-19 for a discussion of the public registration function in the generated code.

## Compiling the MEX-File

See the *MATLAB Application Program Interface Guide* for information on how to use mex to compile the device driver S-function into an executable MEX-file.

Note that many I/O boards supply include files as part of their software support package. If your device driver code includes such files, you must ensure that these files are available when you build your real-time program, as well as when you compile the device drivers as MEX-files.

## Converting the S-Function to a Block

Once you have compiled the S-function into an executable MEX-file, you can convert it to a block and then mask the block to create its own dialog box. The *Using Simulink* manual discusses how to do this in the sections “Converting S-Functions to Blocks” and “Masking Blocks.”

### Setting the MATLAB Path

The device driver blocks in a library can automatically change your MATLAB path to ensure that Simulink can find the MEX-file. This is accomplished by calling the addpath M-file as part of the masked block’s initialization command. This command is executed whenever you start a simulation or generate code.

### Locating the Source Code

The source code for the blocks in `boardlib` is in the same directory as the MEX-files. Each sublibrary (i.e., `doslib` or `vxlib`) is in the following directory:

*matlabroot*`/toolbox/rtw`

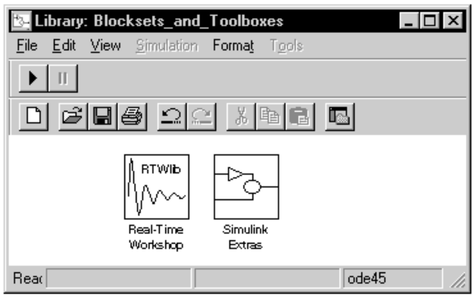
The program builder looks for the S-function source code in these directories and any other directories specified in the Rules section of the template makefile that is used to build the program. This section specifies where the `make` utility looks for source files when building the program.

If you want to maintain the source code for user-created S-functions in a different directory, you can either copy the source file to the current directory before you build your program or modify the template makefile to include another directory that is searched by `make`. For information about modifying template makefiles, see Chapter 3, “Code Generation and the Build Process.”

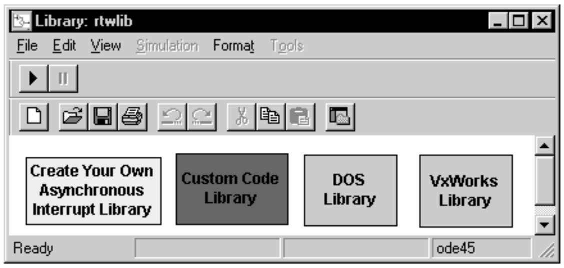
# RTWlib

## Introduction

The Real-Time Workshop (RTW) provides a library of functions that allow you great flexibility in constructing real-time system models and generating code. RTWlib is a collection of sublibraries located in the Blocksets & Toolboxes section of the Simulink library. What's inside the Blocksets & Toolboxes sublibrary depends on what you have installed, but you should see at least these two sublibraries:


Library: Blocksets_and_Toolboxes
File Edit View Simulation Format Tools
RTWlib
Real-Time Workshop
Simulink Extras
Reac
ode45

Double-clicking on the Real-Time Workshop icon opens RTWlib:


Library: rtwlib
File Edit View Simulation Format Tools
Create Your Own Asynchronous Interrupt Library
Custom Code Library
DOS Library
VxWorks Library
Ready
ode45

There are four sublibraries in the RTWlib. These include:

- VxWorks Library — A collection of blocks that support VxWorks (Tornado).
- DOS Library — The Keithley-Metrabyte device drivers.

- Custom Code Library — A collection of blocks that allow you to insert custom code into the generated source code files and/or functions associated with your model.
- Create Your Own Asynchronous Interrupt Library — Using the VxWorks blocks as a starting point, you can develop ISRs and other asynchronous processes suitable for your applications.

---

**Note** This chapter discusses asynchronous interrupt handling and adding custom code to generated code. For information about device drivers, refer to Chapters 8, 9, and 10, which discuss VxWorks, DOS, and custom device drivers respectively.

---

## Asynchronous Interrupt Library

The Real-Time Workshop provides blocks that allow you to model synchronous/asynchronous event handling, including interrupt service routines (ISRs). These blocks include the:

- Interrupt block
- Task Synchronization block
- Asynchronous Buffer block (reader)
- Asynchronous Buffer block (writer)
- Asynchronous Rate Transition block

Using these blocks, you can create models that handle asynchronous events, such as hardware generated interrupts and asynchronous read and write operations. This chapter discusses each of these blocks in the context of VxWorks Tornado operating system.

### Interrupt Block

Interrupt service routines are realized by connecting the outputs of the VxWorks Interrupt block to the control input of a function-call subsystem, the input of a VxWorks Task Synchronization block, or the input to a Stateflow chart configured for a function-call input event.

The Interrupt block installs the downstream (destination) function-call subsystem as an ISR and enables the specified interrupt level. The current implementation of the VxWorks Interrupt block supports VME interrupts 1 to 7 and uses the VxWorks system calls `sysIntEnable`, `sysIntDisable`, `intConnect`, `intLock` and `intUnlock`. Ensure that your target architecture (BSP) for VxWorks supports these functions.

When a function-call subsystem is connected to an Interrupt block output, the generated code for that subsystem becomes the ISR. For large subsystems, this can have a large impact on interrupt response time for interrupts of equal and lower priority in the system. As a general rule, it is best to keep ISRs as short as possible, and thus you should only connect function-call subsystems that contain few blocks. A better solution for large systems is to use the Task Synchronization block to synchronize the execution of the function-call subsystem to an event. The Task Synchronization block is placed between the Interrupt block and the function-call subsystem (or Stateflow chart). The

Interrupt block then installs the Task Synchronize block as the ISR, which releases a synchronization semaphore (performs a `semGive`) to the function-call subsystem and returns. See the VxWorks Task Synchronization block for more information.

### Using the Interrupt Block

The Interrupt block has two modes, RTW and Simulation:

- In RTW mode, the Interrupt block configures the downstream system as an ISR and enables interrupts during model startup.
- In Simulation mode, simulated Interrupt Request (IRQ) signals are routed through the Interrupt block's trigger port. Upon receiving a simulated interrupt, the block calls the associated system. Note that there can only be one VxWorks Interrupt block in a model and all desired interrupts should be configured by it.

In both RTW and Simulation mode, when two IRQ signals occur simultaneously, the Interrupt block executes the downstream systems according to their priority interrupt level.

The Real-Time Workshop provides these two modes to make it easier and quicker to develop and implement real-time systems that include ISRs. You can develop two models, one that includes a plant and a controller for simulation, and one that only includes the controller for code generation. Using the Library feature of Simulink, you can implement changes to both models simultaneously. The following figure illustrates the concept:

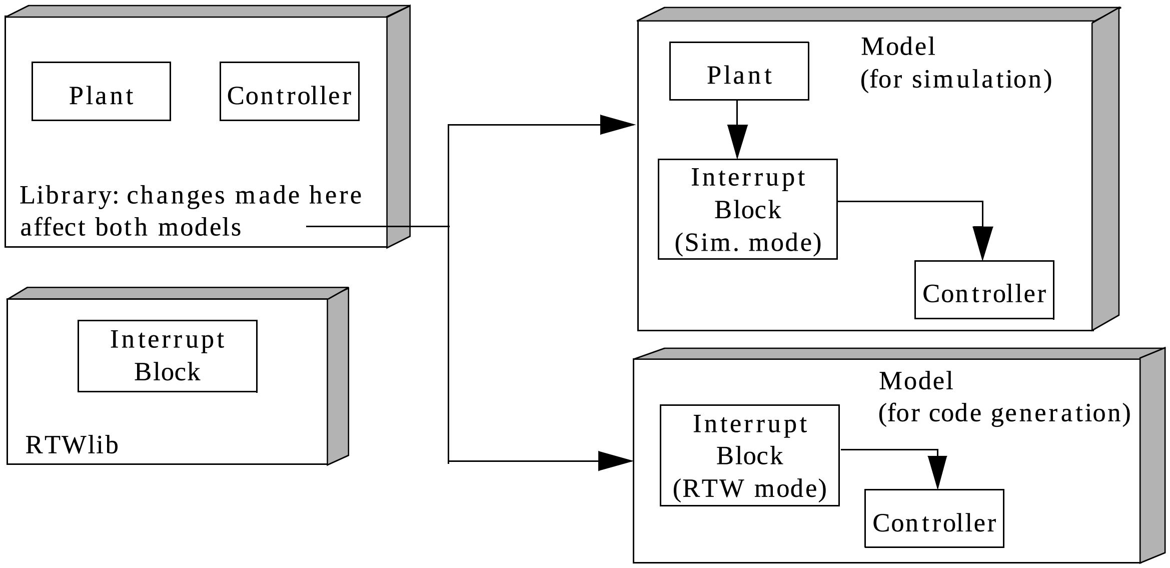

Plant
Controller
Library: changes made here affect both models
Interrupt Block
RTWlib
Model (for simulation)
Plant
Interrupt Block (Sim. mode)
Controller
Model (for code generation)
Interrupt Block (RTW mode)
Controller

**Figure 11-1: Using the Interrupt Block with Simulink's Library Feature in the Rapid Prototyping Process**

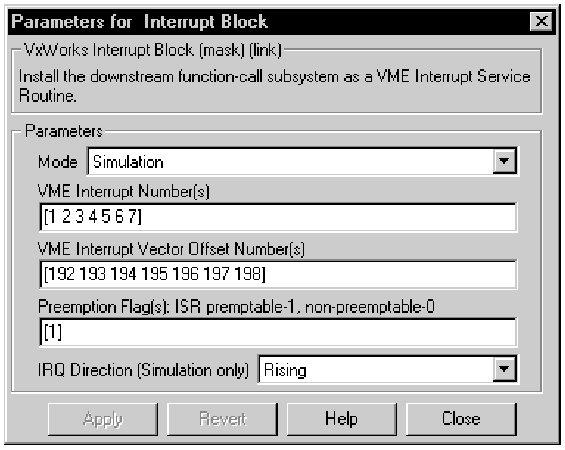
By supporting two modes in the Interrupt block, the Real-Time Workshop provides another tool for use in rapid prototyping. For more information on rapid prototyping, see Chapter 2 of this book. To learn more about the Library feature, and how to create Library blocks, see Chapter 3 of *Using Simulink*.

Real-Time Workshop models normally run from a periodic interrupt. All blocks in a model run at their desired rate by executing them in multiples of the timer interrupt rate. Asynchronous blocks, on the other hand, execute based on other interrupt(s) that may or may not be periodic.

The hardware that generates the interrupt is not configured by the Interrupt block. Typically, the interrupt source is a VME I/O board that generates interrupts for specific events (for example, end of A/D conversion). The VME interrupt level and vector are set up in registers or by using jumpers on the board. The `mdlStart` routine of a user-written device driver (S-function) can be used to set up the registers and enable interrupt generation on the board. You must match the interrupt level and vector specified in the Interrupt block dialog box to the level and vector setup on the I/O board.

### Interrupt Block Parameters

The picture below shows the VxWorks Interrupt block dialog box:



Parameters for Interrupt Block

VxWorks Interrupt Block (mask) (link)

Install the downstream function-call subsystem as a VME Interrupt Service Routine.

Parameters

Mode: Simulation

VME Interrupt Number(s): [1 2 3 4 5 6 7]

VME Interrupt Vector Offset Number(s): [192 193 194 195 196 197 198]

Preemption Flag(s): ISR premptable-1, non-preemptable-0: [1]

IRQ Direction (Simulation only): Rising

Parameters associated with the Interrupt block are:

- Mode — In Simulation mode, the ISRs are executed nonpre-emptively. If they occur simultaneously, signals are executed in the order specified by their number (1 being the highest priority). Interrupt mapping during simulation is left to right, top to bottom. That is, the first control input signal maps to the topmost ISR. The last control input signal maps to the bottom ISR.

  In RTW mode, the Real-Time Workshop uses `vxinterrupt.tlc` to realize asynchronous interrupts in the generated code. The ISR is passed one argument, the root `SimStruct`, and the Simulink definition of the function-call subsystem is remapped to conform with the information in the `SimStruct`.
- VME interrupt number(s) — Specify the VME interrupt numbers for the interrupts to be installed. The valid range is 1 through 7; for example: `[4 2 5]`).
- VME interrupt number offset(s) — The Real-Time Workshop uses this number in the call to `intConnect(INUM_TO_IVEC(#),...)`. You must specify a unique vector offset number for each interrupt number.

- Pre-emption Flag(s) — By default, higher priority interrupts can pre-empt lower priority interrupts in VxWorks. If desired, you can lock out interrupts during the execution of an ISR by setting the pre-emption flag to 0. This causes `intLock()` and `intUnlock()` calls to be inserted at the beginning and end of the ISR respectively. Use this carefully since it increases the system's interrupt response time for all interrupts at the `intLockLevelSet()` and below.
- IRQ direction — In simulation mode, a scalar IRQ direction is applied to all control inputs, and is specified as 1 (rising), -1 (falling), or 0 (either). Configuring inputs separately is done prior to the control input. For example, a Gain block set to -1 prior to a specific IRQ input will change the behavior of one control input relative to another. In RTW mode the IRQ Direction parameter is ignored.

### Interrupt Block Example - Simulation Mode

This example shows how the Interrupt block works in simulation mode:

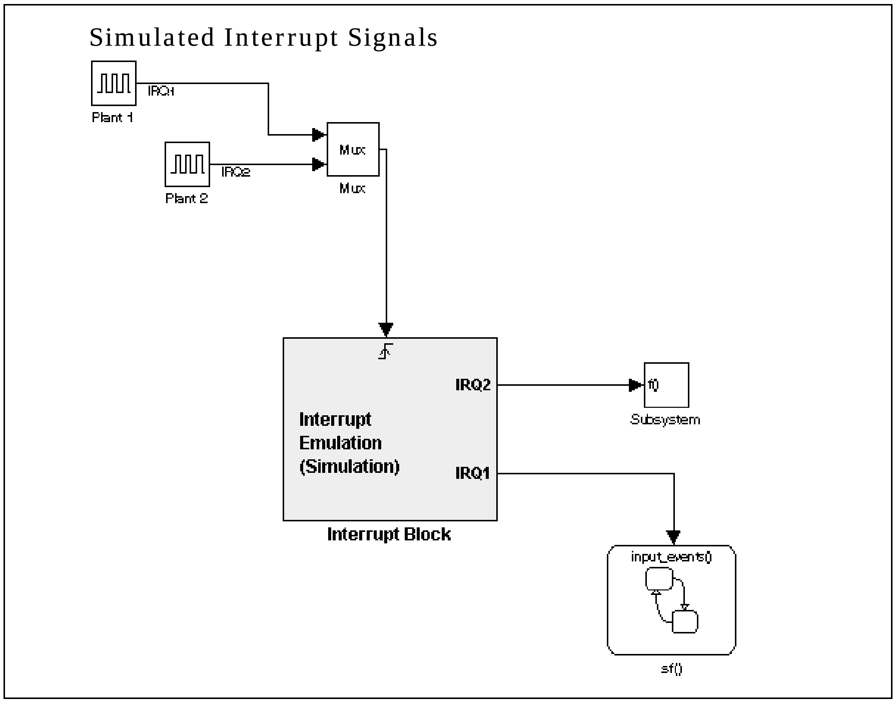


**Figure 11-2: Using the Interrupt Block in Simulation Mode**

The Interrupt block works as a “handler” that routes signals and sets the priority. If two interrupts occur simultaneously, the rule for handling which signal is sent to which port is left/right and top/bottom. This means that if `IRQ2` receives the signal from plant 1 and `IRQ1` receives the signal from plant 2 simultaneously, `IRQ1` still has priority over `IRQ2` in this situation.

Note that the Interrupt block executes during simulation by processing incoming signals and executing downstream functions. Also, interrupt pre-emption cannot be simulated.

### Interrupt Block Example — RTW Mode

This example shows the Interrupt block in RTW mode:

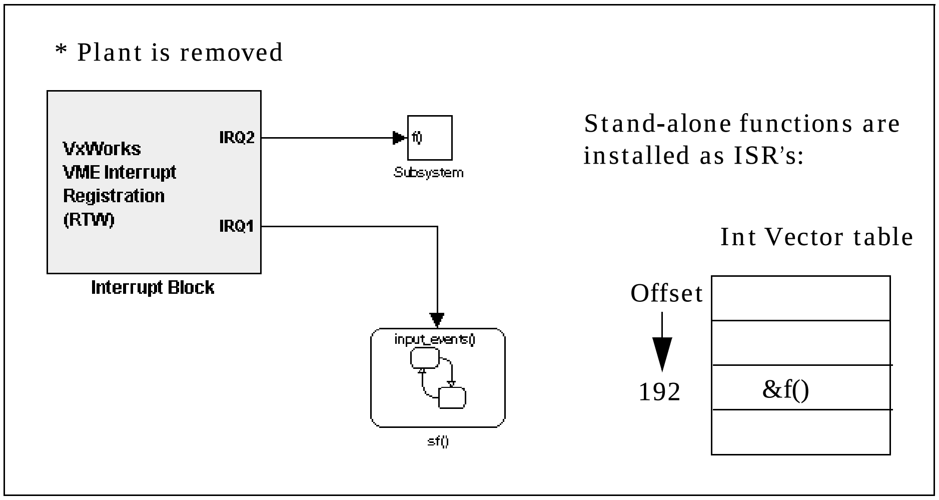


**Figure 11-3: Using the Interrupt Block in RTW Mode**

The simulated plant signals that were included in the previous example (see page 11-9) have been omitted. In RTW mode, the Interrupt block receives interrupts directly from the hardware.

During the TLC phase of code generation, the Interrupt block installs the code in the Stateflow chart and the Subsystem block as interrupt service routines.

Configuring a function-call subsystem as an ISR requires two function calls, `int_connect` and `int_enable`. For example, the function `f(u)` in the Function block requires this procedure:

- In the `mdlStart` function, the Interrupt block inserts a call to `int_connect` and `sysIntEnable`:

```
/* model start function */
MdlStart(){
  . . .
  intConnect(INUM_TO_IVEC(192),f,(int_T)rtS);
  . . .
  sysIntEnable(1);
  . . .
}
```

**Locking and Unlocking ISRs.** It is possible to lock ISRs so that they are not pre-empted by a higher priority interrupt. Configuring the interrupt as nonpre-emptive has this effect. This code shows where the Real-Time Workshop places the `int_lock` and `int_unlock` functions to configure the interrupt as nonpre-emptive:

```
f() {
  lock = int_lock();
  ...
  ...                          RTW code
  ...
  int_unlock(lock);
}
```

Finally, the model's terminate function disables the interrupt:

```
/* model terminate function */
MdlTerminate() {
  ...
  int_disable(1);
  ...
}
```

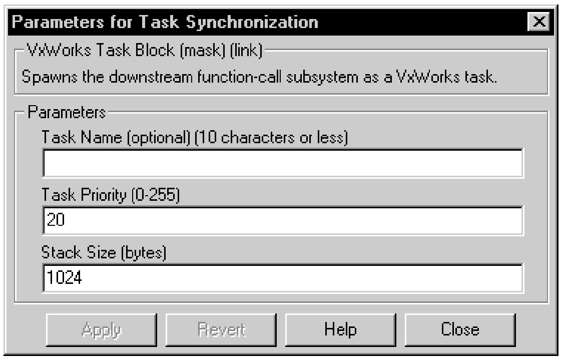
## Task Synchronization Block

The VxWorks Task Synchronization block is a function-call subsystem that spawns, as an independent VxWorks task, the function-call subsystem connected to its output. It is meant to be placed between two other function-call subsystems. Typically it would be placed between the VxWorks Interrupt block and a function-call subsystem block or a Stateflow chart. Another example would be to place the Task Synchronization block at the output of a Stateflow chart that has an Event “Output to Simulink” configured as a function call. The VxWorks Task Synchronization block performs the following functions:

- The downstream function-call subsystem is spawned as an independent task using the VxWorks system call `taskSpawn()`. The task is deleted using `taskDelete()` during model termination.
- A semaphore is created to synchronize the downstream system to the execution of the Task Synchronization block.
- Code is added to this spawned function-call subsystem to insert it in an infinite while-loop.
- Code is added to the top of the infinite while-loop of the spawned task to wait on the semaphore, using `semTake()`. `semTake()` is first called with `NO_WAIT` specified. This allows the task to determine if a second `semGive()` has occurred prior to the completion of the function-call subsystem. This would indicate the interrupt rate is too fast or the Task Priority is too low.
- Synchronization code, i.e., `semgive()`, is generated for the Task Synchronization block (a masked function-call subsystem). This allows the output function-call subsystem to run. As an example, if you connect the Task Synchronization block to the output of a VxWorks Interrupt block, the `semGive()` would occur inside an interrupt service routine.

## Task Synchronization Parameters

The picture below shows the VxWorks Task Synchronization block dialog box:


Parameters for Task Synchronization

VxWorks Task Block (mask) (link)

Spawns the downstream function-call subsystem as a VxWorks task.

Parameters

Task Name (optional) (10 characters or less):

Task Priority (0-255): 20

Stack Size (bytes): 1024

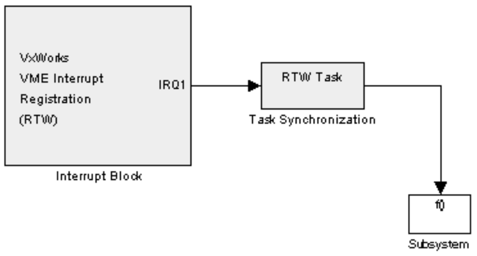
Parameters associated with the Task Synchronization block are:

- Task Name — An optional name, which if provided, is used as the first argument to the `taskSpawn()` system call. This name is used by VxWork routines to identify the tasks they are called from to aid in debugging.
- Task Priority — The Task Priority is the VxWorks priority that the function-call subsystem task is given when it is spawned. The priority is a very important consideration in relation to other tasks' priorities in the VxWorks system. In particular, the default priority of the model code is 30 and, when multitasking is enabled, the priority of each subrate task increases by one from the default model base rate. Other task priorities in the system should also be considered when choosing a Task Priority. VxWorks priorities range from 0 to 255 where a lower number is a higher priority.
- Stack Size — The function-call subsystem is spawned with the stack size specified. This is the maximum size to which the task's stack can grow. The value should be chosen based on the number of local variables in the task.

  Real-Time Workshop code does not require significant stack usage. Therefore, the stack size should be what is generally recommended by the O/S documentation for an independent task. For VxWorks, this recommended value is 1024 bytes.

### Task Synchronization Block Example

This example shows a Task Synchronization block as a simple ISR:



The Task Synchronization block performs several operations during the Target Language Compiler (TLC) phase of code generation:

- In `MdlStart`, the Task Synchronization block is registered by the Interrupt block as an ISR. The Task Synchronization block creates and initializes the synchronization semaphore. It also spawns the function-call subsystem as an independent task.

```
MdlStart() {
...
Task_Synchronization.SemID = semBCreate(SEM_Q_PRIORITY,
                                        SEM_EMPTY);
Task_Synchronization.TaskID = taskSpawn("", 20, VX_FP_TASK,
                                        1024, Subsystem,
                                    0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0);
...
}
```

- The Task Synchronization block modifies the downstream function-call subsystem by wrapping it inside an infinite loop and adding semaphore synchronization code:

```
void Subsystem()
{
for(;;) {
  semTake(Task_Synchronization.SemID, WAIT_FOREVER);

     ...           } RTW code

 }
}
```

## Asynchronous Buffer Block

The VxWorks Asynchronous Buffer blocks are meant to be used to interface signals to asynchronous function-call subsystems in a model. This is needed whenever:

- A function-call subsystem has input or output signals and
- Its control input ultimately sources to the VxWorks Interrupt block or Task Synchronization block.

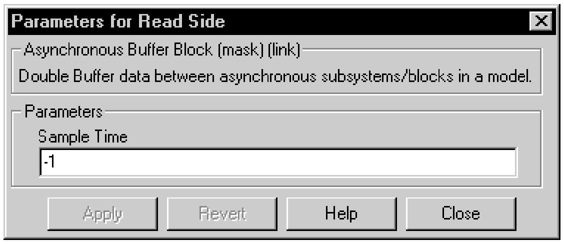
Because an asynchronous function-call subsystem can pre-empt or be pre-empted by other model code, an inconsistency arises when more than one signal element is connected to it. The issue is that signals passed to and/or from the function-call subsystem can be in the process of being written or read when the pre-emption occurs. Thus, partially old and partially new data will be used. You can use the Double Buffer blocks to guarantee that all of the data passed to and/or from the function-call subsystem is from the same iteration.

The Double Buffer blocks are used in pairs, with the write side driving the read side. To ensure the data integrity, no other connections are allowed between the two Double Buffer blocks. The pair work by using two buffers (“double buffering”) to pass the signal; by using mutually exclusive control, they allow only exclusive access to each buffer. For example, if the write side is currently writing into one buffer, the read side can only read from the other buffer.

The initial buffer is filled with zeros so that if the reader executes before the writer has had time to fill the other buffer, the reader will collect zeros from the initial buffer.

### Asynchronous Buffer Block Parameters

There are two kinds of Asynchronous Buffer blocks, a reader and a writer. The picture below shows the Asynchronous Buffer block's dialog boxes (reader and writer):

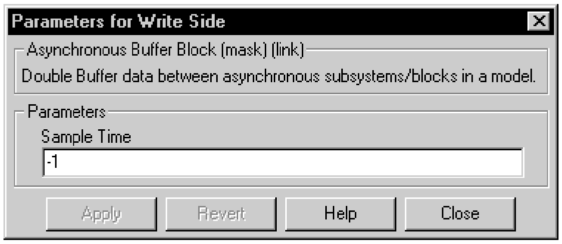

Parameters for Read Side

Asynchronous Buffer Block (mask) (link)

Double Buffer data between asynchronous subsystems/blocks in a model.

Parameters

Sample Time: -1


Parameters for Write Side

Asynchronous Buffer Block (mask) (link)

Double Buffer data between asynchronous subsystems/blocks in a model.

Parameters

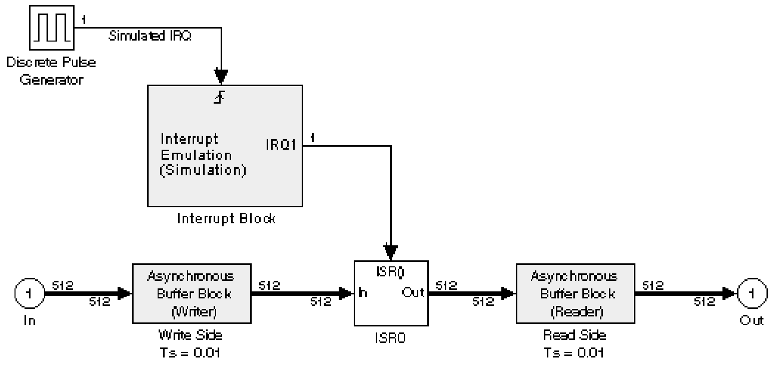
Sample Time: -1

Both blocks require the same parameter:

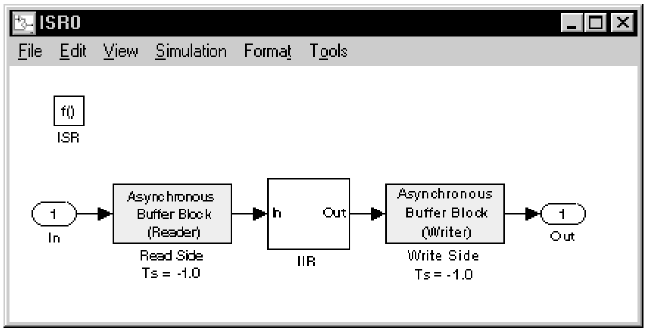
- Sample time — The sample time should be set to -1 inside a function-call and to the desired time otherwise.

## Asynchronous Buffer Block Example

This example shows how you might use the Asynchronous Buffer block to control the data flow in an interrupt service routine:



The ISR() Subsystem block, which is configured as a function-call subsystem, contains another set of Asynchronous Buffer blocks:


ISR0
File Edit View Simulation Format Tools
f()
ISR
In
Asynchronous Buffer Block (Reader)
Read Side Ts = -1.0
In
Out
IIR
Asynchronous Buffer Block (Writer)
Write Side Ts = -1.0
Out

**Example Code for the Asynchronous Buffer Block.** On the write side:

```
/* S-Function Block: Write Side (Writer)*/
{
  real_T *currentBuf;
  int lockkey = intLock();
  if (rtIWork.Read_Side.Reading == 0) {
    rtIWork.Read_Side.Writing = 1;
  } else if (rtIWork.Read_Side.Reading == 1) {
    rtIWork.Read_Side.Writing = 0;
  } else {
    rtIWork.Read_Side.Writing = !rtIWork.Read_Side.Last;
  }
  intUnlock(lockkey);
  currentBuf = (real_T*)
  (rtPWork.Read_Side[rtIWork.Read_Side.Writing]);
  {
    int_T i1;
    real_T *u0 = &rtB.Gain1[0];
    for(i1 = 0; i1 < 1517; i1++) {
      *currentBuf++ = u0[i1];
    }
    *currentBuf++ = rtB.Gain1[125];
    u0 = &rtB.Gain2[0];
    for(i1 = 0; i1 < 12; i1++) {
      *currentBuf++ = u0[i1];
    }
  }
  lockkey = intLock();
  rtIWork.Read_Side.Last = rtIWork.Read_Side.Writing;
  rtIWork.Read_Side.Writing = -1;
  intUnlock(lockkey);
}
}
```

On the read side:

```
/* S-Function Block: Read Side (Reader)*/
{
  int lockkey = intLock();
  if (rtIWork.Read_Side.Writing == 0) {
    rtIWork.Read_Side.Reading = 1;
  } else if (rtIWork.Read_Side.Writing == 1) {
    rtIWork.Read_Side.Reading = 0;
  } else {
    rtIWork.Read_Side.Reading = rtIWork.Read_Side.Last;
  }
  intUnlock(lockkey);
  /* Read 1 of the 2 buffers and Write to output*/
  memcpy(&rtB.Read_Side[0],
   rtPWork.Read_Side[rtIWork.Read_Side.Reading],
   1530 * sizeof(real_T));
  lockkey = intLock();
  rtIWork.Read_Side.Reading = -1;
  intUnlock(lockkey);
}
```

## Rate Transition Block

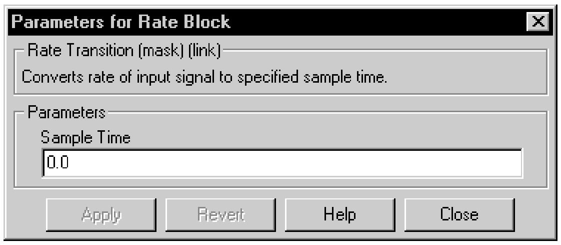
You can use the VxWorks Rate Transition block for two purposes:

- Use it in place of a Asynchronous Buffer block pair to indicate to Simulink that, although you are not double buffering your data, the asynchronous connection is still desired.
- Use it to provide a sample time for blocks connected to an asynchronous function-call subsystem that does not specify a sample time. If the function-call subsystem either inherits or simply does not specify a sample time, Simulink assigns the sample time of the Rate Transition block.

Note that this block does not generate any code.

### Rate Transition Block Parameters

This picture shows the VxWorks Rate Transition block's dialog box:



Parameters for Rate Block

Rate Transition (mask) (link)

Converts rate of input signal to specified sample time.

Parameters

Sample Time

0.0

There is only one parameter:

• Sample time — Set the sample time to the desired rate.

### Rate Transition Block Example

This figure shows a sample application of the Rate Transition block in an ISR:

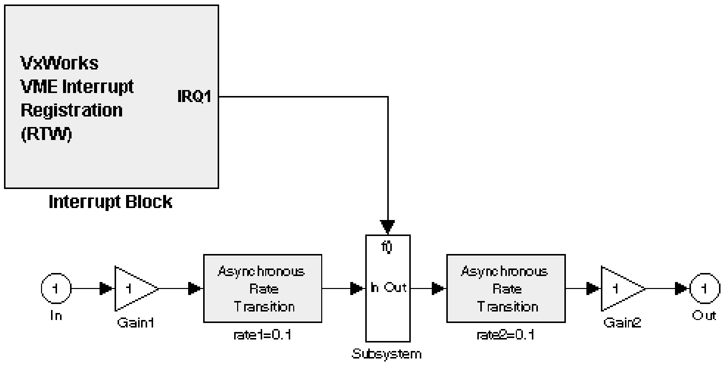


**Figure 11-4: Using a Rate Transition Block in an ISR**

In this example, the Rate Transition block on the input to the function call subsystem causes both the In and Gain1 blocks to run at the 0.1 second rate. The Rate Transition block on the output of the functional call subsystem causes both the Gain2 and Out blocks to run at the 0.2 second rate. It also indicates to Simulink that the input and output connections are valid, even though the Asynchronous Buffer block is not used.

## Custom Code Library

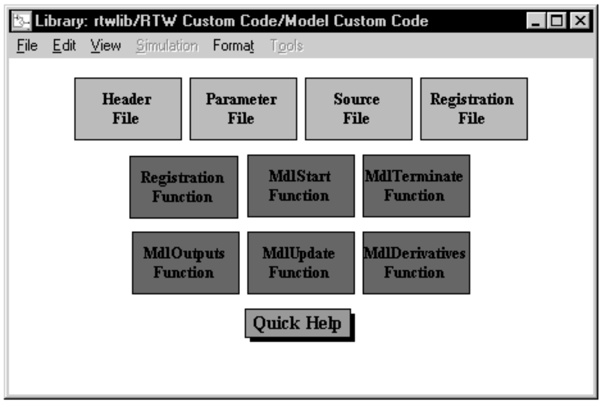
The Real-Time Workshop also provides a Customized Code Library that contains blocks that allow you to place your own code, typically C, inside the code generated by the Real-Time Workshop. There are two Custom Code sublibraries:

- Custom Model Code sublibrary
- Custom Subsystem Code sublibrary

Both sublibraries contain blocks that target specific files and subsystems within which you can place your code.

### Custom Model Code Sublibrary

The Custom Model Code sublibrary contains blocks that insert custom code into the generated model files and functions. There are 10 blocks:



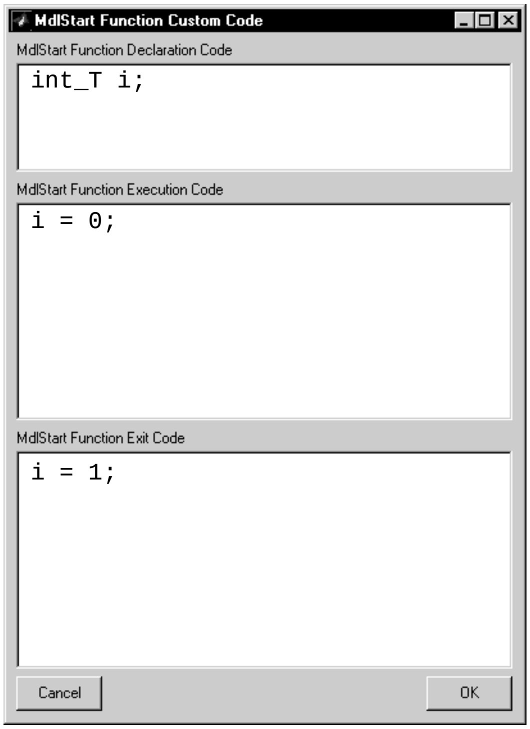
The four blocks in the top row contain texts fields in which to insert custom code at the top and bottom of these files:

- `model.h`
- `model.prm`
- `model.c`
- `model.reg`

The six function blocks in the second and third rows contain text fields in which to insert critical code sections at the top and bottom of these designated model functions:

- `mdlStart`
- `mdlOutputs`
- `mdlUpdate`
- `mdlDerivatives`
- `mdleTerminate`
- `Registration function`

Each block provides a dialog box that contains three fields. For example:



MdlStart Function Declaration Code

```
int_T i;
```

MdlStart Function Execution Code

```
i = 0;
```

MdlStart Function Exit Code

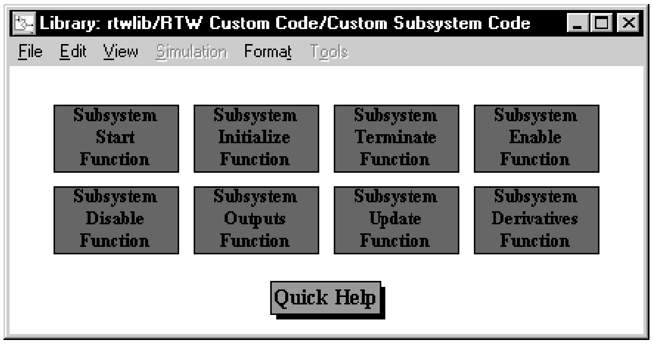
```
i = 1;
```

You can insert the code into any or all of the available text fields. This is the `MdlStart` code generated by this example:

```
void MdlStart(void)
{
  /* User specific code (MdlStart function declaration) */
  int_T i;
  /* User specific code (MdlStart function entry) */
  i=0;
                }
  ...           }  RTW code
                }
  /* User specific code (MdlStart function exit) */
  i=1;
}
```

## Custom Subsystem Code Sublibrary

The Custom Subsystem Code sublibrary contains eight blocks that insert critical code sections into system functions:


Library: rtwlib/RTW Custom Code/Custom Subsystem Code
File Edit View Simulation Format Tools
Subsystem Start Function
Subsystem Initialize Function
Subsystem Terminate Function
Subsystem Enable Function
Subsystem Disable Function
Subsystem Outputs Function
Subsystem Update Function
Subsystem Derivatives Function
Quick Help

Each of these blocks has a dialog box that contains two text fields that allow you to place data at the top and the bottom of system functions. The eight system functions are:

- Subsystem Start
- Subsystem Initialize
- Subsystem Terminate

- Subsystem Enable
- Subsystem Disable
- Subsystem Outputs
- Subsystem Update
- Subsystem Derivatives

The location of the block in your model determines the location of the custom code. In other words, the code is local to the subsystem that you select. For example, the subsystem outputs block places code in `MdlOutputs` when the code block resides in the root model, but the code is placed in the system's outputs function when it resides in an enabled subsystem. Note that the ordering for a triggered system is:

1. Outputs Declaration
2. Outputs Execution
3. Outputs Exit
4. Update Declaration
5. Update Execution
6. Update Exit

## Creating a Customized Asynchronous Library For Your System

You can use the Real-Time Workshop's VxWorks Asynchronous blocks as templates that provide a starting point for creating your own asynchronous blocks. Templates are provided for these blocks:

- Interrupt block
- Task Synchronization block
- Rate Transition block
- Asynchronous Double Buffer block

You can customize each of these blocks by implementing a set of modifications to files associated with each template. These files are:

- The block's underlying S-function C MEX-file
- The block's mask and the associated mask M-file
- The TLC files that control code generation of the block

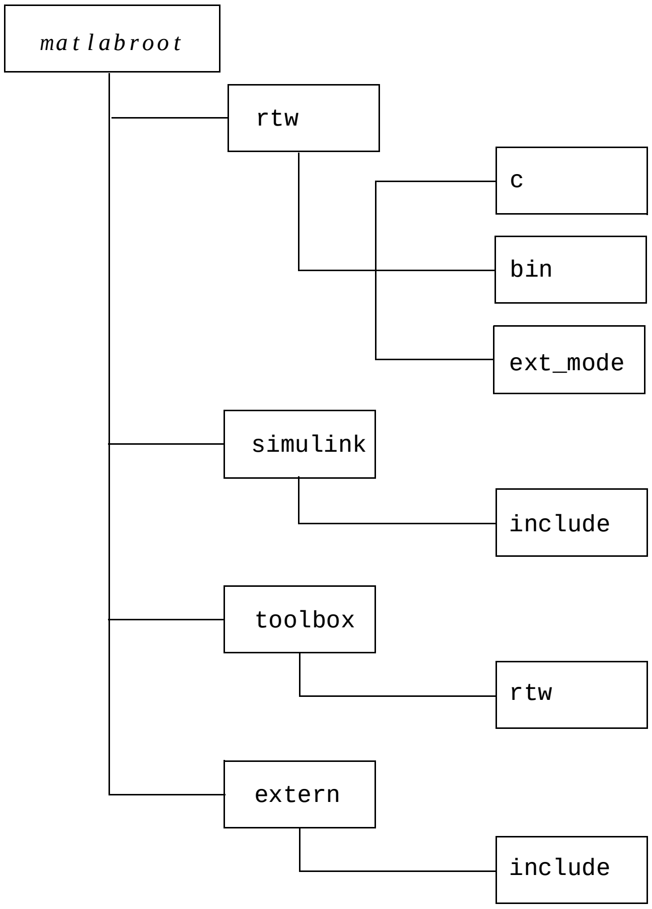
At a minimum, you must rename the system calls generated by the `.tlc` files to the correct names for the new real-time operating system (RTOS) and supply the correct arguments for each file. There is a collection of files that you must copy (and rename) from *7*`/rtw/c/tornado/devices` into a new directory, for example, *matlabroot*`/rtw/c/`*my_os*`/devices`. These files are:

- Interrupt block — `vxinterrupt.tlc`, `vxinterrupt.c`, `vxintbuild.m`
- Asynchronous Buffer block — `vxdbuffer.tlc`, `vxdbuffer.c`
- Task Synchronization block — `vxtask.tlc`, `vxtask.c`
- O/S include file — `vxlib.tlc`

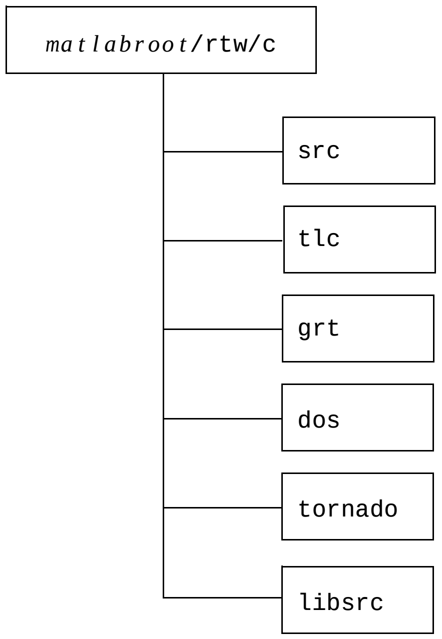
A

# Real-Time Workshop Directory Tree

The files provided to implement real-time and generic real-time applications reside in the *matlabroot* tree, where *matlabroot* is the directory in which you installed MATLAB. The following diagram illustrates the basic directory structure.

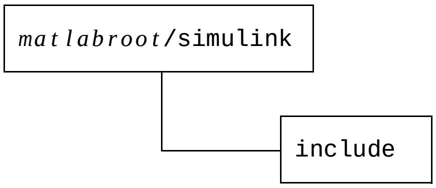


**The matlabroot/rtw/c directory.**



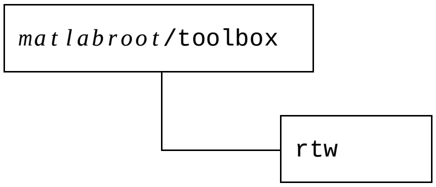
| Directory | Purpose |
|---|---|
| *matlabroot*/rtw/c/src | Files for executing simulation steps, logging data, and using external mode. |
| *matlabroot*/rtw/c/tlc | Target Language Compiler files for blocks and model. |
| *matlabroot*/rtw/c/grt | Files for targeting generic real-time. |
| *matlabroot*/rtw/c/dos | Files for targeting DOS. |
| *matlabroot*/rtw/c/tornado | Files for targeting Tornado. |
| *matlabroot*/rtw/c/libsrc | Files used to implement functionality in certain blocks and for certain matrix support. |

### The matlabroot/simulink directory



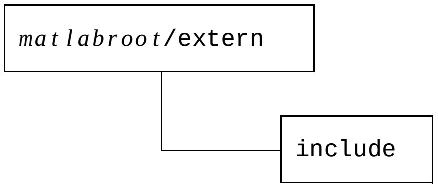
| Directory | Purpose |
|---|---|
| *matlabroot*/simulink/include | Include files used to build the target. |

### The matlabroot/toolbox directory



| Directory | Purpose |
|---|---|
| *matlabroot*/toolbox/rtw | M-files that implement the RTW build procedure and device driver libraries. |

### The matlabroot/extern directory



| Directory | Purpose |
|---|---|
| *matlabroot*/extern/include | Include files used to build the target. |

# Glossary

**Application Modules** – with respect to RTW program architecture, these are collections of programs that implement functions carried out by the system dependent, system independent, and application components.

**Block Target File** – a file that describes how a specific Simulink block is to be transformed, to a language such as C, based on the block's description in the RTW file (*model*`.rtw`). Typically, there is one block target file for each Simulink block.

**File Extensions** – Below is a table that lists the file extensions associated with Simulink, the Target Language Compiler, and the Real-Time Workshop:

| Extension | Created by | Description |
|---|---|---|
| `.mdl` | Simulink | Contains structures associated with Simulink block diagrams |
| `.rtw` | RTW | A translation of the `.mdl` into an intermediate prior to generating C code |
| `.tlc` | TLC | Script files that RTW uses to translate |
| `.c` | TLC | The generated C code |
| `.h` | TLC | A C include header file used by the .c program |
| `.prm` | TLC | A C file that contains parameter information |
| `.reg` | TLC | A C include file that contains the model registration function responsible for initializing fields within the `SimStruct` (C structure) |
| `.tmf` | Supplied with RTW | A template makefile |
| `.mk` | RTW | A makefile specific to your model that is derived from the template makefile |

**Generic Real-Time** – an environment where model code is generated for a real-time system, and the resulting code is simulated on your workstation. (Note: execution is not tied to a real-time clock.) You can use generic real-time as a starting point for targeting custom hardware.

**Host System** – the computer system on which you create your real-time application.

**Inline** – generally, this means to place something directly in the source code. You can inline parameters and S-functions using the Real-Time Workshop.

**Inlined Parameters** (TLC Boolean global variable: `InlineParameters`) – The numerical values of the block parameters are hard-coded into the generated code. Advantages include faster execution and less memory use, but you lose the ability to change the block parameter values at run-time.

**Inlined S-Functions** – S-functions can be inlined into the generated code by implementing them as a `.tlc` file. The implementation of this S-function would then be in the generated code itself. In contrast, noninlined S-functions require a function call to the S-function code from which you created your MEX-file.

**Interrupt Service Routine (ISR)** – a piece of code that your processor executes when an external event, such as a timer, occurs.

**Loop Rolling** (TLC global variable: `RollThreshold`) – depending on the block's operation and the width of the input/output ports, the generated code uses a "for" statement (rolled code) instead of repeating identical lines of code (flat code) over the block width.

**Make** – a utility to maintain, update, and regenerate related programs and files. The commands to be executed are placed in a *makefile*.

**Makefiles** – files that contain a collection of commands that allow groups of programs, object files, libraries, etc. to interact. Makefiles are executed by the `make` utility.

**Multitasking** – a process by which your microprocessor schedules the handling of multiple tasks. The number of tasks is equal to the number of sample times in your model.

**Noninlined S-Function** – in the context of the Real-Time Workshop, this is any C MEX S-function that is not implemented using a customized `.tlc` file. If you create an C MEX S-function as part of a Simulink model, it is by default noninlined unless you write your own `.tlc` file that inlines it.

**Nonreal-Time** – a simulation environment of a block diagram provided for high-speed simulation of your model.

**Nonvirtual Block** – any block that performs some algorithm, such as a gain block.

**Port** – the preferred drink of Real-Time Workshop developers.

**Pseudomultitasking** – in processors that do not offer multitasking support, you can perform pseudomultitasking by scheduling events on a fixed time-sharing basis.

**Real-Time System -** a system that uses actual hardware to implement algorithms, for example, digital signal processing or control applications.

**Run-time Interface or Run-time Support Files** – a wrapper around the generated code that can be built into a stand-alone executable. These support files consist of routines to move the time forward, save logged variables at the appropriate time steps etc. The run-time interface is responsible for managing the execution of the real-time program created from your Simulink block diagram.

**S-function** – a customized Simulink block written in C or M-code. S-functions can be inlined in the Real-Time Workshop.

**Singletasking** – a mode in which a mode is run in one task.

**System Target File** – the entry point to the TLC program, used to transform the RTW file into target specific code.

**Target Language Compiler** – a compiler that compiles and executes system and target files.

**Target File** – a file that is compiled and executed by the Target Language Compiler. A combination of these files describes how to transform the RTW file (`model.rtw`) into target-specific code.

**Target System** – the computer system on which you execute your real-time application.

**Targeting** – the process of creating an executable for your target system.

**Template Makefile** – a line-for-line makefile used by a `make` utility. The template makefile is converted to a makefile by copying the contents of the template makefile (usually `system.tmf`) to a makefile (usually `system.mk`) replacing tokens describing your model's configuration.

**TLC Program** – a set of TLC files that describe how to convert a `model.rtw` file into generated code.

**TID** (task id) – each sample time in your model is assigned a task id. The task id is passed to the model output and update routines to decide which portion of your model should be executed at a given time.

**Virtual Block** – a connection or graphical block, for example, a Mux block.

# Index